Code Quality Improvement for All: Automated Refactoring for Scratch

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

***Abstract*—Block-based programming has been overwhelmingly successful in revitalizing introductory computing education and in facilitating end-user development. However, poor code quality makes block-based programs hard to understand, modify, and reuse, thus hurting the educational and productivity effectiveness of blocks. There is great potential benefit in empowering programmers in this domain to systematically improve the code quality of their projects. Refactoring—improving code quality while preserving its semantics—has been widely adopted in traditional software development. In this work, we introduce refactoring to Scratch. We define four new Scratch refactorings: Extract Custom Block, Extract Parent Sprite, Extract Constant, and Reduce Variable Scope. To automate the application of these refactorings, we enhance the Scratch programming environment with powerful program analysis and transformation routines. To evaluate the utility of these refactorings, we apply them to remove the code smells detected in a representative dataset of 448 Scratch projects. We also conduct a between-subjects user study with 24 participants to assess how our refactoring tools impact programmers. Our results show that refactoring improves the subjects' code quality metrics, while our refactoring tools help motivate programmers to improve code quality.**

***Index Terms*—block-based languages, software refactoring, Scratch, code quality, code smells, program analysis, end-user programming, introductory curriculum**

I. Introduction

Block-based programming plays an essential role in realizing the vision of *CS for All* [1], which renders computing accessible to the broadest possible audience of programmers, many of whom are learners and end-users. A highly popular block-based programming language is Scratch, whose general design principles strive for "a low-barrier to entry" for beginners and "a high-ceiling" for maturing programmers to create increasingly sophisticated programs over time [2]. Nevertheless, the Scratch community's main focus thus far has been on its "low-barrier to entry," with new command blocks that accommodate a wider audience by rendering programming accessible and attractive. Perhaps as an unintended consequence, the efforts to push the "ceiling" higher have been somewhat deprioritized. Left to their own devices, experienced programmers do get to work on advanced sophisticated projects, but at the cost of their software growing uncontrollably in size and complexity, with the overall code quality becoming degraded over time.

As it turns out, the issues of code quality reduce the pedagogical effectiveness of block-based programming [3], [4] as well as the attractiveness of blocks for serious end-user programming pursuits [5]. Consequently, this programming community can no longer afford to neglect the issues of code quality and its systematic improvement. To that end, support for improving code quality can play an important role in elevating the "ceiling" of block-based programming, while also transforming code quality improvement into a regular practice of novice programmers and end-user developers.

As first steps on the way to improving quality, several recent research efforts have focused on identifying recurring quality problems in block-based software [6], [4]. However, once faced with the identified quality problems, a programmer needs to decide whether to spend the time and effort required to fix them. In fact, evidence suggests novice programmers in this domain refrain from engaging in quality improvement practices [7], despite being sufficiently proficient in programming to improve their code. Across all levels of expertise, novice programmers have been observed introducing a high number of quality problems in their code [8].

Integrating quality improvement practices into the software development process can be prohibitively expensive for professional software developers, let alone novice programmers and end-users. For text-based languages, automated refactoring has become an indispensable quality improvement tool [9]. Refactoring systematically improves code quality, while keeping its functionality intact, ensured by its sophisticated program analyses and behavior-preserving program transformations. Alas, major block-based programming environments only support the most rudimentary `Rename` refactoring, which removes the *Uncommunicative Name* code smell [4]. However, other highly recurring quality problems (e.g., code duplication) make block-based projects hard to understand, modify, and reuse. To be able to improve the code quality of their projects, programmers need well-documented refactorings for Scratch and programming support, which both identifies refactoring opportunities and applies them automatically. Systematically supporting Scratch programmers in improving the code quality of their projects can increase the pedagogical and productivity benefits of this programming domain.

To address this problem, we added automated refactoring support to Scratch 3.0, validated by implementing four refactorings that remove code smells, reported as *highly recurring*

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

in prior studies [6], [4]. Extract Custom Block puts the repeated sequences of statement blocks into a new procedure, invoked in place of the sequences. Extract Parent Sprite removes duplicate sprites (programmable objects) by introducing a special construct that clones a sprite multiple times. Extract Constant replaces recurring constant expressions with a new variable, initialized to that expression. Reduce Variable Scope changes a variable scope accessibility from all sprites (default setting) to a given sprite.

With the exception of Extract Custom Block, these refactorings would be novel even without automated application. They provide the Scratch community with a vocabulary to communicate about semantic preserving transformations that improve code quality. These refactorings are also uniquely applicable to Scratch, as it is this language's domain-specific features and distinct programming practices that cause the code smells these refactorings remove. Extract Custom Block does have counterparts in text-based languages, but to correctly carry out this refactoring in Scratch requires verifying a different set of correctness preconditions.

To determine the potential usefulness of the introduced refactorings, we apply them to a representative sample of 448 Scratch projects in the public domain. We ran a user study to investigate whether the availability of our refactoring impact the studied participants improving code quality and their opinion about code quality and our refactoring tools.

The evaluation results show that overall (3 out of 4 refactorings) show the high applicability of 79% or higher, when applied to the code smells previously shown to be highly prevalent in the Scratch codebase. Each refactoring positively impacts its respective software metrics, which improve various code quality attributes, including program size, comprehensibility, modifiability, and abstraction. Our user study results suggest that the presence of actionable improvement hints and the associated refactorings motivates programmers to improve code quality. To the best of our knowledge, this work is the first effort to introduce automated refactoring to Scratch. By describing the design, implementation, and evaluation of our approach, this paper makes the following contributions:

1) A catalog of four refactorings for Scratch that removes highly recurring code smells.
2) An intuitive user interface for refactoring, whose actionable and contextualized coding hints encourage programmers to engage in improving code quality.
3) An experimental study that evaluates the applicability and utility of the introduced refactorings.
4) A user study that investigates the impact of our refactoring tools on programmers.
5) A software architecture and reference implementation of a refactoring engine for Scratch 3.0, featuring program analyzers and automatic code transformation.

The rest of the paper is structured as follows. Section II provides the technical background of this research and compares this work to the related state of the art. Section III presents a catalog of Scratch refactorings. Section IV describes our automated software refactoring support. Section V evaluates the introduced refactorings. Section VI discusses the significance of the evaluation results. Section VII presents concluding remarks and future work directions.

II. Background and Related Work

This section provides the background information required to understand our contributions and also reviews the most closely related prior research efforts.

A. Block-Based Programming Languages

By lowering the barrier to entry for beginner programmers, block-based programming has achieved an unprecedented level of success and popularity [10]. The two areas in which block-based programming has been most successful are introductory computing education and end-user programming. In this domain, programming environments typically separate the front-end blocks editor from the back-end execution engine, with this separation enabling different languages to reuse the same blocks editor.

Our analysis and infrastructure targets Scratch [2]. To support automated refactoring, we enhance the latest version of Scratch, whose blocks editing interface is built on Blockly [11], a popular blocks editing framework. With minimally built-in semantic analyses capabilities, the Blockly framework leaves it up to language designers to implement the necessary semantic analyses capabilities. Refactoring relies on program analyses, whose program representation differs from the blocks editor's representation, designed for rendering and interactively manipulating blocks.

B. Automated Software Refactoring

Automated software refactoring has received considerable attention from the research community [12]. We review the most closely related examples of prior work, from which we draw lessons and insights required to design and implement automated refactoring support for block-based languages.

a) Analyses and Transformations: Several facets of our refactoring infrastructure build upon the prior advances in automated refactoring despite its text-based context. Refactoring engines commonly operates on the representation known as *a program graph* [13], an AST augmented with semantics edges that express various relationships (e.g., reference binding, def-use chains, etc.). We adopt this representation for its flexibility in analyses and transformations, in which additional semantics edges are introduced as required by a given analysis. For example, control flow and data flow analyses in Scratch require the information about broadcast-receive relationship in the program. To that end, we leverage JastAdd, a Java-based language processing framework by Hedin and Eva [14]. Our analyses follow the design of the AST-level extensible intraprocedural program analysis by Söderberg et al. [15].

b) Refactoring for Blocks: Despite its ubiquitous availability in the IDEs for text-based languages, refactoring has only been scarcely applied in the domains of end-user programming (e.g., pipe-like mashups [16], spreadsheet [17]). In block-based programming, the Blockly framework provides rudimentary refactoring support: renaming variables and changing function signatures (i.e., add parameters and change their order). These built-in refactorings can be implemented by following a simple match-and-replace program transformation strategy, insufficient to support our advanced refactorings.

Several prior works analyze code quality [6], [4] and create analysis tools (i.e., Hairball [18], Dr.Scratch [19], and Quality Hound [20]). However, these prior works neither focus on how to address quality problems nor on how to apply automated tools. By identifying recurring Scratch code smells, these prior works inspire the refactorings described herein.

We designed our refactoring user interface to favor simplicity over versatility, in line with our target audience. That is, a quality hint is presented as a light bulb, on which the programmer can mouse over to see the suggested code improvement context, and then right-click to apply the suggested refactoring. We chose this simple design over more complex interfaces, such as those that embrace the native drag-and-drop idioms of block-based programming. Even though these idioms inspired a novel refactoring user interface for Java [21], they would be poorly suited for refactoring block-based code, as they require a detailed knowledge of the source and target destinations of the intended refactoring transformations. Hence, drag-and-drop refactoring interfaces are better suited for object hierarchies, with clearly delineated boundaries between program constructs. Our user study answers the fundamental question of whether access to automated refactoring support motivates novice programmers and end-users to improve code quality in this domain.

III. Refactoring Catalog

Next we present our refactoring catalog. For each refactoring, we list its rationale and preconditions. Due to space limitation, we only illustrate the affected program parts before and after the application of complex refactorings.

A. Extract Custom Block

This refactoring creates a procedure, whose body comprises the repeated code fragment being refactored, and replaces all occurrences of this fragment with the appropriately parameterized invocations of the created procedure.

Precondition: For behavioral preserving transformation, each argument to be parameterized must be a constant and can be parameterized. Note that some blocks accept a drop-down option value and cannot be parameterized. Finally, the extracted fragment must not contain the control flow terminating command (i.e., "`stop <this script>`" block).

B. Extract Parent Sprite

This refactoring removes duplicate sprites by extracting the parent sprite which instantiates its children clones using "`create clone of <target>`" block[1]. Encapsulated within a sprite, the same code is not only easier to modify, but is also amenable to other localized refactorings (e.g., Extract Custom Block, Extract Constant, etc.). Automated refactoring cannot remove sprite duplicates in all cases. Some of the removals require adding boilerplate code, which would be hard to generate automatically and require advanced programming expertise to understand. Hence, the refactoring presented herein is applicable when sprite duplicates share similar code (usually at the beginning when a duplicate has just been introduced). Note that the "hide" block is immediately executed in the parent sprite, so as to emulate an invisible prototype object, whose only purpose is to clone visible children.

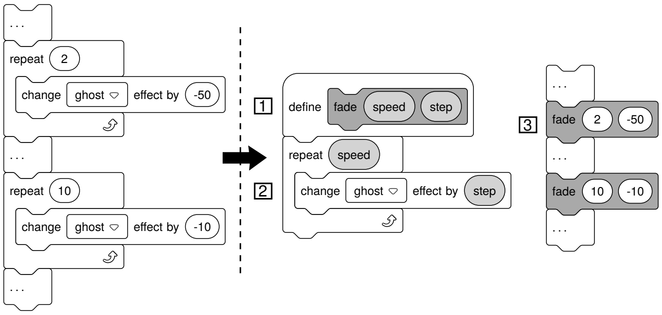


Fig. 1. Extract Custom Block

Preconditions: Sprite duplicates have identical set of scripts (exact code duplication, without variation in literals and identifiers (e.g. variable references)). Each sprite duplicate contains no scripts starting with the "`when I start as a clone`" block. Finally, this Each sprite duplicate uses a single costume.

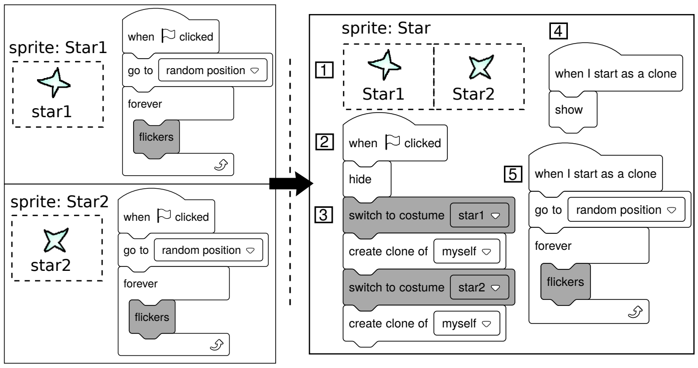


Fig. 2. Extract Parent Sprite

C. Extract Constant

This refactoring replaces replicated constant values with a variable. Descriptively named variables improve comprehension and modifiability [22]. The only *precondition* is that the replicated values must be of type literal.

D. Reduce Variable Scope

This refactoring changes the scope of an existing variable from being accessible to all sprites to only a given sprite. If global scope for a variable is not needed, reducing its scope improves the sprite's data encapsulation.

Preconditions: Only one sprite modifies the rescoped variable (though it can be read by multiple sprites)

[1] https://en.scratch-wiki.info/wiki/Cloning

IV. Refactoring for Scratch

Although we introduce automated refactoring to Scratch, our general architecture and design can be applied to any block-based programming environment.

Overall Architecture: Exposed as remote services, the required program analysis and transformation functionalities integrate non-intrusively. Passed a serialized form of the edited program as input, these services analyze and detect code smells, returning the computed refactoring transformations.

Implementation: Based on its input parameters, the *refactoring engine* analyzes and transforms the edited program. The refactoring parameters can be specified by the programmer or in our case automatically extracted from the smells (e.g., Duplicate Code → Extract Custom Block request). Before performing any transformations, the *refactoring engine* determines whether a given refactoring request satisfies all of its preconditions. In the transformation phase, the *refactoring engine* modifies the analysis AST, while recording each modification as a *transformation action*. Having been transferred back to the client-side, this atomic sequence of actions is applied to the *program model*, maintained by the block-based programming environment. The actions are applied in the specified order, as each of them modifies program state.

Fig.3 shows some transformation action types used to implement Extract Constant. Each action is persisted, so the client can replay the corresponding transformations on the client-side's *program model*. Our design assumes all program elements (both blocks and non-blocks) can be looked up based on their string IDs, so program changes can be mapped across representations. Additionally, the blocks editor can serialize and deserialize its internal *program model* (e.g., in XML or JSON data format).

Our program analysis and transformation operate on an AST by means of JastAdd [14], a language processing framework previously mentioned in II. We express various relationships between program elements in a *program graph* with its declarative specification language to augment the AST classes.

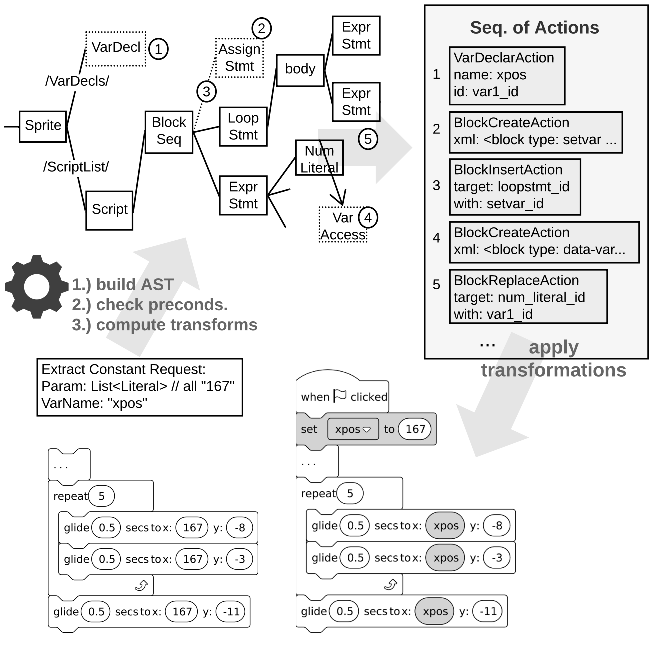


Fig. 3. Different stages of Extract Constant refactoring

Fig.3 illustrates the major phases of refactoring with an example of performing Extract Constant. The first phase starts with a refactoring request, whose parameters for Extract Constant comprise all the block's IDs of all duplicate literals and the edited program. To determine if all preconditions are met, the server-side refactoring engine executes various analysis routines (e.g., `check preconds`) on the parsed AST. Then, the engine computes and record a sequence of transformations (i.e., "Actions") that put the refactoring into effect ("compute transforms"). The resulting transformation actions are serialized and returned to the blocks editor, which presents the discovered smell hints along with the suggested refactoring transformations ("apply transformations").

Refactoring Interface Design: While experienced programmers eagerly refactor their code, novice programmers are unfamiliar with the practice. Hence, the latter's willingness to refactor needs to be encouraged with a friendly and intuitive user interface. Refactoring starts from identifying code whose quality can be improved, a hard task that is even harder for novice programmers. To render refactoring accessible to our target audience, we follow two key design principles, also demonstrated in the screenshot in Fig.4, an example of applying Extract Custom Block refactoring to a real-world Scratch project.

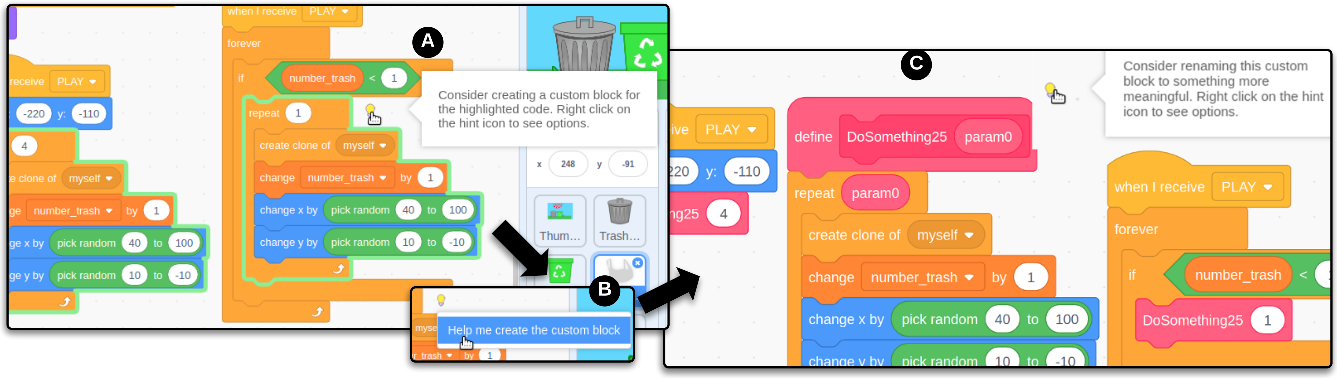
1) *Code smells should be presented as improvement opportunities to the programmer*. Fig.4A displays a code hint as a light-bulb icon, indicating an opportunity for improving code quality (Extract Custom Block in this case). Whenever possible a hint should be visually contextualized. For Extract Custom Block refactoring, our refactoring interface highlights duplicate code blocks.
2) *Refactoring should be immediately actionable*. Instead of relying on the programmer to specify the required refactoring parameters, as in traditional refactoring, the infrastructure should present only the actions ready for the programmer to act upon. Fig.4B shows *"Help me create the custom block"*, the only available action for this hint in a simple terminology that can be easily understood by novice and end-user programmers.

Note that in this example, an additional refactoring hint, shown after the application of Extract Custom Block, suggests to the programmer that the just-extracted custom block should be meaningfully renamed.

V. Evaluation

Automated refactoring can become helpful for novice and end-user programmers in improving the quality of their projects, as long as the refactorings are applicable, useful, and accessible for this programming audience. Our evaluation seeks answers to the following questions:

RQ1. How applicable is each introduced refactoring?
RQ2. How do the refactorings impact code quality?



Consider creating a custom block for the highlighted code. Right click on the hint icon to see options.

Help me create the custom block

Consider renaming this custom block to something more meaningful. Right click on the hint icon to see options.

Fig. 4. A screenshot of the EXTRACT CUSTOM BLOCK refactoring invocation interface for Scratch

RQ3. Do refactoring tools motivate quality improvement?

To answer **RQ1** and **RQ2**, we experimentally evaluate a representative sample of Scratch projects. We refactor the code smells, automatically detected by our infrastructure's smell analysis modules. To answer **RQ3**, we conduct an online between-subjects study, in which participants in the treatment group interact with our refactoring infrastructure and answer survey questions.

A. *Experimental Evaluation*

We limit the scope of our evaluation to highly prevalent smells and whether our refactorings can remove them. Note that some refactorings can remove more than one type of code smell (e.g., EXTRACT CUSTOM BLOCK can remove both LONG SCRIPT [4] and DUPLICATE CODE smells). Hence, if we were to apply the available refactorings to remove all automatically detected smells, such an evaluation strategy would distort the applicability results and the refactored code quality, as some of the detected smells may not be indicative of actual quality problems (e.g.,what constitutes a LONG SCRIPT is highly subjective). To avoid such distortions, our evaluation considers the following fixed SMELL→REFACTORING pairs: DUPLICATE CODE→EXTRACT CUSTOM BLOCK, DUPLICATE SPRITE→EXTRACT PARENT SPRITE, DUPLICATE CONSTANT→EXTRACT CONSTANT, and BROAD SCOPE VARIABLE→REDUCE VARIABLE SCOPE.

Evaluation Dataset: To assess how viable the refactorings are, we measure the applicability and impact of applying them to third-party Scratch programs. An API request[2] to MIT's Scratch service retrieves a list of projects, divided into two categories of approximately equal size: (1) trending and (2) recent. This subject selection strategy ensures that we conduct our evaluation on a diverse sample of projects created by the Scratch community. We collected a total of 448 projects 51% among them were viewed at most once, with the rest of projects were viewed on average 12,749 times. Among the subject projects, 88% were remixed at most once and the rest were remixed on average 93 times.

Code Smell	Definition and Detection Criteria
Duplicate Code	2 or more code fragments, containing more than one statement, are duplicate if they have identical structure except for variations in identifiers and literals (type II in clone classification [23]). If multiple duplicate fragments overlap, the largest is selected.
Duplicate Sprite	2 or more sprites are duplicate if each script within one of the sprites is duplicated in the others.
Duplicate Constant	Exact literals of at least 3 characters that are replicated at least twice (the thresholds identified experimentally to reduce false positive results)
Broad Scope Variable	A variable declared in the global scope (Stage), but assigned only locally in a single sprite

TABLE I
CODE SMELL DEFINITIONS

RQ1. Refactoring Applicability: For each refactoring, we assess its applicability by calculating the percent of its associated smells that are refactorable. Because code smell definitions affect the applicability of refactorings, Table I lists the considered smells and their detection criteria as the bases for interpreting our evaluation results.

Smell → Refactoring	Afflicted Projects	Total Smells	Refactored Smells
Duplicate Code → Extract Custom Block	181 (41%)	290	229 (79%)
Duplicate Sprite → Extract Parent Sprite	142 (32%)	193	22 (11%)
Duplicate Constant → Extract Constant	194 (43%)	453	453 (100%)
Broad Scope Var. → Reduce Var. Scope	94 (21%)	145	118 (81%)

TABLE II
APPLICABILITY (N=448)

Results: Table II summarizes the results of evaluating refactoring applicability. In our evaluation, as long as a project contains at least one instance of a given code smell, the project is considered *afflicted* by that smell. Different smells have been found to afflict different projects in the evaluation dataset.

Afflicting over 30% of the subject projects, duplication-related smells are the most prevalent; afflicting around 21%, BROAD SCOPE VARIABLE is the least prevalent smell. One project may contain more than one instance of the same code smell (Total.Smells > No.Afflicted.Projects). We use all detected smells to evaluate refactoring applicability.

[2] https://scratch.mit.edu/explore/projects/all/<recent>|<trending>

Metric	Definition
LOC	# statement blocks within a program
Complex Script Dens.	% of scripts (including procedure) with McCabe's cyclomatic complexity [24] value $>$ 10 (risk threshold according to [25])
Long Script Dens.	% of scripts (including procedure) with *LOC* $>$ 11 LOC (threshold empirically determined in previous work [4])
Procedure Dens.	# procedures within a program per 100 *LOCs*
No. Literals	# literals (numbers and strings) within a program
No. Global Var.	# global variables
No. Create Clone Of.	# CREATECLONEOF<TARGET> blocks

TABLE III
METRICS DEFINITIONS

The applicability of the introduced refactorings varies widely. With the success rate of over 75%, EXTRACT CONSTANT, REDUCE VARIABLE SCOPE, and EXTRACT CUSTOM BLOCK are the most applicable refactorings. EXTRACT CUSTOM BLOCK's precondition failures are due to the variations in duplicate fragments failing to satisfy the preconditions (60% of the variations contain global variables and 31% contain non-constant expression blocks; 9% are located at non-parameterizable input slots). As expected, EXTRACT PARENT SPRITE is the least applicable refactoring due to its restrictive preconditions—only 11% of the detected smell instances can be refactored. The reason for failures is that EXTRACT PARENT SPRITE cannot handle certain duplicate sprites, out of which 63% differ slightly in terms of their contained code, 33% are multi-costumes, and 4% contain scripts starting with the "`when I start as a clone`" block. Overall, we observe the introduced refactorings to be satisfactorily applicable to the highly recurring smell types. Even the least applicable refactoring—EXTRACT PARENT SPRITE—can be applied frequently enough.

RQ2. Refactoring Impact on Quality: To assess how each refactoring impacts program quality, we apply all the evaluated refactorings in sequence on each of the detected smell type instances. We then compute the relevant software metrics of the original and the refactored versions of each subject program, so as to determine the difference or the delta in code quality, which serves as a measure of refactoring quality impact. Table III defines the software metrics used.

Results: Table IV summarizes the characteristics of the detected smells that are refactorable. Table V summarizes the percentage changes for different software metrics before and after performing each refactoring. To help the reader interpret the results, the last column translates the mean deltas into percent improvements. *Group Size* refers to the number of replications of a given program element. Next, we describe the results in terms of different code quality attributes. *Total Uses* refers to the number of times a given variable is read in a project. *External Uses* refers to the number of times a given variable is read from outside the sprite in which it is defined.

Size: We expect duplication-eliminating refactorings to remove redundant code and decrease the code size of the afflicted projects. We observe that EXTRACT CUSTOM BLOCK reduces a varying level of code size. A small improvement in code size is due to the small number of repetitions detected more frequently than bigger ones. Thus, most projects see a mean decrease in LOC by 3.38%. Though less applicable, EXTRACT PARENT SPRITE refactoring removes large duplications at the sprite level. A subset of projects afflicted by DUPLICATE SPRITE see a greater mean decrease in LOC by 8.49%.

Metrics	N	Min	p25	Med	Mean	p75	Max
Duplicate Code							
Group Size	229	2	2	3	3.05	3	20
Fragment Size	229	3	3	4	5.05	6	27
Duplicate Sprite							
Group Size	22	2	2	2.5	22.86	4.75	238
Sprite Size (LOC)	22	1	1	1.0	1.64	2.00	3
Duplicate Constant							
Group Size	453	5	5	6	8.96	10	113
Literal Length	453	2	2	3	3.14	3	58
Broad Scope Variable							
Total Uses	118	0	1	1	2.73	2.00	60
External Uses	90	0	0	0	0.47	0.75	6

TABLE IV
CHARACTERISTICS OF REFACTORABLE SMELLS

Comprehension: We expect EXTRACT CUSTOM BLOCK to help shorten some long scripts and reduce the number of complex scripts due to the original script being extracted. We observe 4.4% decrease in the number of long scripts. On the other hand, we only observe a slight improvement in terms of the reduction in the number of complex scripts (5.77%) indicating that most refactorable Duplicate Code smells are *not* located within complex scripts.

Modifiability: Although the software metrics literature still lacks a definitive metrics known to faithfully capture code modifiability, we can still reason about certain code modifiability improvements by measuring the number of repeated functionalities that have become localized in a single reusable program unit (i.e., procedure for DUPLICATE CODE, parent sprite for DUPLICATE SPRITE, and variable for DUPLICATE CONSTANT). In Table IV, the *Group Size* characteristic of these refactored duplication-related smells reflects the number of locations a programmer needs to navigate to make similar changes in the duplicate parts.

Abstraction: Duplication-eliminating refactorings have an obvious impact on abstraction (i.e., EXTRACT CUSTOM BLOCK increases procedural abstraction, EXTRACT PARENT SPRITE increases object abstraction, and EXTRACT CONSTANT increases uses of variable, a basic data abstraction). Lastly, REDUCE VARIABLE SCOPE improves information hiding or encapsulation, which correlates with the increase in the usage of local variables. The result indicates the refactored projects (N=41) which have used local variables at least once could see an increased usage of local variables by almost 54% on average. A great room for improvement in the usages of local variable is expected as changing the scope of declared variable in Scratch is an expensive and tedious transformation requiring the programmer to create a new variable with the intended scope and replace each existing variable block with the newly created one manually.

B. User Study

We conducted an online user study, facilitated by Amazon's Mechanical Turk, in order to gain access to a diverse pool of

	% Change Statistics							% Improve
Metrics	N	Min	p25	Med	Mean	p75	Max	
Extract Custom Block								
LOC	147	-29.70	-4.16	-1.65	-3.38	-0.67	0.00	+3.38%
Complex Script Dens.	52	-100.00	-4.00	-2.47	-5.77	-1.44	-0.28	+5.77%
Long Script Dens.	137	-100.00	-6.45	0.66	-4.40	1.97	42.24	+4.4%
Procedure Dens.	46	2.72	13.61	35.48	49.98	54.73	222.15	+49.98%
Extract Parent Sprite								
LOC	20	-94.15	-12.71	-0.69	-8.49	4.45	41.18	+8.49%
No. Sprites	20	-96.39	-58.48	-12.70	-31.07	-8.90	-3.85	+31.07%
Extract Constant								
No. Literals	194	-65	-14.02	-8.33	-11.53	-4.7	-0.89	+11.53%
Reduce Variable Scope								
No. Local Variable	43	2.22	7.28	20	52.42	66.67	200	+52.42%

TABLE V

Percentage Changes of Software Metrics Before and After Refactorings

novice and experienced participants. A total of 24 participants took part in the study. 7 out of 13 participants in the treatment group reported having programming experience as compared to 4 out of 11 in the control group. The participants took 30 minutes on average to complete the study (1-hour hard limit) and were compensated $3 for completing the assignment. This study investigated the impact of the availability of refactoring tools (i.e., code quality hints and automatic refactorings) on the propensity of programmers to improve the quality of their code. To that end, the participants were first primed to use custom blocks to improve program comprehensibility, and then encouraged to improve their code, amenable to the Extract Custom Block refactoring.

The participants first answered background questions about their programming experience and familiarity with Scratch. Then they received a short introduction to Scratch programming and custom blocks. To prime the participants to think about code quality, they were asked to rank two program versions on their comprehensibility (both performed the same animation but one was a refactored version of the other). The participants were presented with a programming task that required reusing in two places an existing block sequence in the workspace. In order to understand what the block sequence did, it was expected to be run. Manually extracting a parameter-less custom block from this code sequence took 5 editing steps. The participants were asked to make sure their code was easy to understand before completing the task. In the remainder of the study, the control and treatment groups diverged. Only the treatment group was exposed to Duplicate Code hints and the associated Extract Custom Block refactoring.

RQ3. Refactoring Tools Motivating Quality Improvement: To investigate how the treatment affected the likelihood of the participants improving code quality, we instrumented our custom Scratch editor to record the following two program versions: 1) the first submission attempt, before participants were asked to make their code easy for others to understand; and 2) the final submission. To understand how the presence of refactoring tools impacted the participants' attitude toward code quality, we asked them to complete a post-study survey.

Results: The study results for each question are:

RQ3.1 Engagement with improving code quality: We tested whether the participants, who chose to engage in improving code quality, depended on receiving our improvement hints and their suggested refactorings. To that end, we performed a Chi-square independence test. The relationship between these variables turned significant, (χ^2 (1, N = 24) = 8.48, P=.004), thus implying that programmers receiving hints were likely to follow them in applying the suggested refactorings.

When asked which of the program versions they found easier to understand, 25% of the participants chose the original version, while 75% of them chose the refactored one. We looked further if the participants' preference for their choices affected their engagement in improving code quality. Among the participants who chose the "refactored version" as being easier to understand, only 12.5% in the control group ended up improving the code quality, as compared to 80% of the participants in the treatment group.

RQ3.2 Code quality perception: When asked whether their finished programs would be easy to understand for novice programmers, 85% of the participants in the treatment group agreed as compared to 91% in the control group.

RQ3.3 Improvement hints and refactoring usefulness: The treatment group was asked how useful they found the improvement hints and the associated refactorings in making their code easy for others to understand. The vast majority of the group members found our refactoring tools useful: 54% *very useful* and 38% *extremely useful*.

C. Threats to Validity

Our study had several threats to validity. Our experimental evaluation results only reflected the partial applicability and quality impact of each refactoring to the studied code smells. As mentioned, some of these refactorings were applicable in additional code smells/scenarios, but not all of them could be properly covered in one study. Because performing a refactoring is a subjective decision, the results did not necessarily equate with the actual applicability and quality impact. Nevertheless, one can see from our results the potential usefulness of providing such support for the programmers in this domain.

In our user study, the participants with some programming experience, but no familiarity with Scratch represented almost a half of the total participants. Although we intended to include more results from novice participants, it would be impossible for us to make use of their incomplete task results. Although, as expected, programming experience was positively correlated with completion rates, it showed no influence on our intervention. The programming task used in the study was not representative of the real world programs in this domain. However, it was reasonably complex for a half of the participants that had no programming experience.

VI. Discussion

Overall, the results of our experimental evaluation and user study are quite revealing. The introduced refactorings do improve the code quality metrics by removing recurring code smells from the representative projects. We have also observed that the presence of improvement hints and their associated refactorings positively influences how programmers perceive code quality and its systematic improvement.

Suggesting Quality Improvements: Although automated hints help detect quality problems, some of the detected quality problems may not need to be refactored due to their triviality. A better alternative can be to provide hints in the form of before and after examples. Other detected problems make poor subjects for automated refactoring due to their high complexity. Indeed, fixing some of these problems may require significant programmer involvement (e.g., to come up with a meaningful name). Finally, some problems are simply not amenable to automated refactoring due to the complexity of formalizing the required general transformation strategies. Nevertheless, some non-trivial refactorings would likely to present a cognitive burden and may interrupt the creative flow, without seamless and effective automated support to encourage their application.

Perceived Code Quality Judging code quality remains somewhat subjective as our user study results reveal. The participants regardless of their programming experience perceive code quality (program comprehensibility) differently. The participants also rate their code positively on how easy it is to be understood by other programmers new to the language, even though their finished programs exhibit a similar quality to the programs they previously ranked as being harder to understand. Receiving no suggestions, programmers may be unaware about all the different alternatives they have to achieve their goal. The availability of automated refactoring influences programmers to become aware of code quality and how they can improve it.

Educational Benefits: These improvement hints and their associated refactorings can provide a timely intervention to help novice programmers become aware of alternative design and implementation options that can improve code quality. For example, certain quality hints and refactorings may alleviate the low usage of procedures (a well-documented observation in a prior work [26]), thus elevating the role of procedural abstraction—a fundamental concept in CS education and professional software development—in this domain. Our evaluation results focusing on the participants without programming experience are very encouraging. Custom blocks or procedures are considered somewhat a hard concept not introduced until later in the introductory curriculum. However, most of the non-programmer participants in the treatment group were able to take advantage of our refactoring tools and perceive the hints and their suggested code improvement actions positively.

VII. Conclusion and Future Work

This paper describes our effort to introduce automated refactoring to Scratch, a widely used block-based programming language. To demonstrate the practicality of our analysis and transformation infrastructure, we implement four refactorings that remove highly recurring Scratch code smells, identified in prior works. By providing their rationales, preconditions and transformation strategy, we systematically document these Scratch refactorings for use by both programmers and language designers. To assess the potential usefulness of each introduced refactoring, we experimentally evaluate the applicability and quality impact of each refactoring on a dataset of 448 projects. Our evaluation results show that the introduced refactorings are highly applicable, while their application improves code quality.

Our refactoring infrastructure helps overcome two main hindrances: programmers being unaware of code improvement opportunities and the programming burden of the improvements. Our infrastructure provides coding hints with immediately actionable suggestions to carry out the refactorings. Our user study reveals that the presence of improvement hints and associated automatic refactorings increases the likelihood of programmers deciding to improve code quality.

Although this work focuses on Scratch, our experiences and findings can benefit designers and developers of other block-based programming environments. Future designs of block-based environments can be improved to make program analyses and transformations easily accessible, so as to facilitate the development of semantic editing support, in addition to improving code quality. Our findings serve as a starting point in determining which refactorings are likely to be useful and worthwhile to programmers in this domain. We plan to investigate further how novice programmers interact with the refactoring tools as part of the overall programming process. The following research questions arise: *How the presence of refactoring tools affects how novice and end-user programmers code? How effectively this presence raises the code quality awareness among programmers?* The answers could inform the research community how much providing refactoring support raises the importance of code quality in the minds of programmers in this increasingly important domain.

Acknowledgments

The authors would like to thank Franklyn Turbak and 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] M. Smith, "Computer science for all," 2016. [Online]. Available: https://www.whitehouse.gov/blog/2016/01/30/computer-science-all

[2] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman *et al.*, "Scratch: programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[3] 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.

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

[5] Y. Ohshima, J. Mnig, and J. Maloney, "A module system for a general-purpose blocks language," in *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, Oct 2015, pp. 39–44.

[6] 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.

[7] 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.

[8] P. Techapalokul and E. Tilevich, "Novice programmers and software quality: Trends and implications," in *2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE&T)*, Nov 2017, pp. 246–250.

[9] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *2009 IEEE 31st International Conference on Software Engineering*, pp. 287–297, 2009.

[10] D. Bau, J. Gray, C. Kelleher, J. Sheldon, and F. Turbak, "Learnable programming: Blocks and beyond," *Commun. ACM*, vol. 60, no. 6, pp. 72–80, May 2017. [Online]. Available: http://doi.acm.org/10.1145/3015455

[11] N. Fraser *et al.*, "Blockly: A visual programming editor," *URL: https://developers.google.com/blockly/*, 2013.

[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.

[13] J. L. Overbey and R. E. Johnson, "Differential precondition checking: A lightweight, reusable analysis for refactoring tools," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, Nov 2011, pp. 303–312.

[14] G. Hedin and E. Magnusson, "JastAdd—an aspect-oriented compiler construction system," *Science of Computer Programming*, vol. 47, no. 1, pp. 37–58, 2003.

[15] E. Söderberg, T. Ekman, G. Hedin, and E. Magnusson, "Extensible intraprocedural flow analysis at the abstract syntax tree level," *Science of Computer Programming*, vol. 78, no. 10, pp. 1809 – 1827, 2013, special section on Language Descriptions Tools and Applications (LDTA'08 & '09) & Special section on Software Engineering Aspects of Ubiquitous Computing and Ambient Intelligence (UCAmI 2011).

[16] K. T. Stolee and S. Elbaum, "Refactoring pipe-like mashups for end-user programmers," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 81–90.

[17] S. Badame and D. Dig, "Refactoring meets spreadsheet formulas," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, Sept 2012, pp. 399–409.

[18] 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*. ACM, 2013, pp. 215–220.

[19] J. Moreno-León, G. Robles, and M. Román-González, "Dr. Scratch: Automatic analysis of Scratch projects to assess and foster computational thinking," *RED. Revista de Educación a Distancia*, no. 46, pp. 1–23, 2015.

[20] P. Techapalokul and E. Tilevich, "Quality Hound — an online code smell analyzer for Scratch programs," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct 2017, pp. 337–338.

[21] Y. Y. Lee, N. Chen, and R. E. Johnson, "Drag-and-drop refactoring: Intuitive and efficient program transformation," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 23–32.

[22] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[23] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470 – 495, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642309000367

[24] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, pp. 308–320, Dec 1976.

[25] M. Bray, K. Brune, D. A. Fisher, J. Foreman, and M. Gerken, "C4 software technology reference guide-a prototype." Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, Tech. Rep., 1997.

[26] I. Li, F. Turbak, and E. Mustafaraj, "Calls of the wild: Exploring procedural abstraction in app inventor," in *2017 IEEE Blocks and Beyond Workshop (B&B)*, Oct 2017, pp. 79–86.