Understanding Recurring Quality Problems and Their Impact on Code Sharing in Block-Based Software

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

***Abstract*—Block-based programming languages have become increasingly prominent in both the educational and end-user communities. As the block-based codebase is growing rapidly, its quality remains poorly understood, even though the awareness of recurring quality problems in this domain can benefit educators and end-user programmers alike. To address this problem, we report on the results of a large-scale assessment of recurring quality problems in block-based software. Our work identifies quality problems endemic of block-based software, as well as applies program analysis to assess the prevalence and severity of quality problems in close to 600K representative Scratch projects. Our empirical evidence shows how certain recurring quality problems hinder code sharing for popular Scratch projects. These results indicate that the quality of block-based software warrants the attention of CS educators, end-user programmers, and tool builders. Our study's results can help programmers avoid introducing the quality problems, while guiding tools builders in supporting the systematic quality improvement of block-based software.**

***Index Terms*—end-user programming; empirical study; software quality; program analysis; code smells; block-based software; Scratch; software refactoring**

I. Introduction

The key functions of modern society critically depend on software-based systems. Finance, transportation, communication, government, defense—all rely on software to manage and carry out day-to-day operations. A key factor that determines the utility and safety of any software-based system is software quality. In this respect, poor software quality is known not only to increase the development and maintenance costs, but also be conducive to causing software defects [1], [2]. Block-based languages have become an important entryway to the world of software development for CS learners and end-user programmers alike. Although one may argue that block-based programs are too simple to warrant any quality concerns, the issue at hand is the formation of good habits that promote solid software engineering practices, as block-based programmers move forward in their computing journeys. In any case, software quality is known to be inversely correlated with the effort required to understand, modify, and evolve a software system [3], [4]. In that light, improving software quality is an important process, with the assessment of quality problems being the critical first step in this process.

In addition to the societal impact, poor software quality can hinder learning enabled by code sharing, an important learning activity for novice programmers. For example, Scratch, the language we focus on here, has a large and engaging online community, whose slogan is "Imagine, Program, Share." This slogan reflects a vision of sharing—called *remixing* in Scratch—being a central tenet of this learning community [5]. Yet, the majority of Scratch projects we studied have had a limited success in allowing others to extend them, rendering these projects less "remixable." As we have discovered, software quality can be an important factor affecting whether a project is remixable.

In this work, we document recurring quality problems in block-based programs written in Scratch by leveraging the well-recognized software quality assessment methodology of *code smells* [6]. In essence, a code smell documents a recurring pattern of design and/or implementation choices that indicate the symptoms of software quality problems. We study how the presence of code smells affects remixed projects in terms of their "remixability." In fact, some code smells that we studied have shown statistically significant effects on how remixable a project is. The intuition behind this insight is simple: for a project to be inviting for other programmers to remix and extend, it has to be easy to understand and modify, a property hindered by the presence of some code smells.

To establish a practical benchmark for the thresholds at which the presence of code smells starts hindering remixing, our study focuses on popular remixed projects, whose remixes have been substantially extended. We then use these benchmark-based thresholds to determine the severity of the discovered code smells in our subject dataset, which comprises close to 600K projects.

The low-risk category of smell severity indicates the level of quality at which a project is likely to be remixed and extended. As our results show, some code smells with high prevalence have a larger percentage of projects in the very high risk category compared to the benchmark projects. Conversely, successful remixed projects exemplify how high software quality can help uphold the fundamental sharing principle of the learning community of novice programmers.

978-1-5386-0443-4/17/$31.00

The goal of our study is to answer the following research questions:

- *RQ1: What known code smells described in the literature are applicable in the context of Scratch?*
  Motivation: The incidence of code smells is commonly influenced by certain programming language features. Although block-based languages share many similarities, they tend to differ with respect to their feature-sets. As a result, recurring quality problems afflict programs in these languages in dissimilar ways. This work contributes a catalog of code smells specific to Scratch, thus far not thoroughly compiled and documented, benefiting all the stakeholders in Scratch and giving insights to the stakeholders in other block-based languages.
- *RQ2: What can the analysis of popular Scratch projects teach us about the state of software quality in this programming domain?*
  Motivation: Popular projects can strongly impact the software development practices of novice programmers, who commonly remix these projects, thus using them as an active learning resource. Understanding the software quality of popular projects can provide insights about how quality affects code remixing and also make it possible to derive practical software quality benchmarks.
- *RQ3: How prevalent and severe are different code smells in the general Scratch code base?*
  Motivation: Knowing how prevalent each type of code smell is in a large population of Scratch programs can provide helpful hints for the tool builders, whose aim is to enable block-based developers to improve the quality of their projects. When providing refactoring tools support, one should pay special attention to the code smells with high prevalence and severity.

This paper makes the following contributions:

1) **A catalog of code smells for Scratch** We present a catalog of code smells for Scratch, drawn from the research literature on recurring software quality problems.
2) **An assessment of how the presence and density of code smells affect the likelihood of a popular Scratch project being remixed and extended** We propose the *Script Addition* metric as a simple heuristic to identify Scratch projects that are likely to exhibit high comprehensibility and extensibility. We statistically evaluate whether the proposed metric can be relied on to predict the rate of incidence of code smells.
3) **A large-scale assessment of the prevalence and severity of code smells in Scratch projects** We present our findings on the prevalence and severity of code smells in a large dataset of close to 600K Scratch projects by using the percentile-based risk thresholds derived from the subset of popular projects whose *Script Addition* metric is ranked in the top 25%.

Paper Organization: The remainder of the paper is organized as follows: Section II presents background information on the Scratch language. Section III compares this work with the related state of the art. Section IV presents our approach to cataloging smells, and the identified catalog as well as explains how we designed our quality assessment study. Section V explains the results of our study. Section VI points out the threats to validity of our results. Finally, Section VII presents future work direction and concluding remarks.

II. Background

This section gives an overview of block-based programming languages and the characteristics of Scratch, the language we focus on in this study.

A. Block-based programming languages

In recent years, block-based programming languages have become increasingly important both in educational pursuits and the computational empowerment of non-CS professionals. The appeal of block-based languages ranges from the pedagogical effectiveness, with which they can introduce computing concepts to introductory computing learners, to the intuitiveness, with which they can represent complex domain-specific computing environments for areas including robotics, multimedia computation, and mobile computing.

B. Scratch

Scratch [7] allows programmers to apply computing concepts to create a wide range of interactive and media rich projects such as games, animation, and storytelling. The Scratch language is specifically designed to support multimedia computing (e.g., interactive stories, games, animations, etc.).

Each Scratch project contains one stage object that acts as the global scope for the whole project. The stage can contain zero or more graphic objects called *sprites*. Both the Stage and sprite objects can be programmed; they are commonly referred to collectively as *scriptables*. A scriptable comprises a collection of scripts as well as a set of multimedia elements (i.e., graphics called costumes, and sounds) that can be referenced by the scripts. Each script is a unit of functionality composed of a series of connected blocks. Hence, a block is a key Scratch construct.

Scratch is an event-based programming language. Each script starts with a special block that is triggered by an event and acts as an entry point of the script. Events comprise user events (e.g.,"a mouse click") and internal events that are created by programmers using a broadcast/receive mechanism used for inter-script communication. Variables can be declared to be globally accessible or private to each sprite. Programmers can create custom blocks which are similar to functions or methods. However, custom blocks cannot be shared between scriptables, and they cannot return values.

III. Related Work

In this section, we describe the research work on code smells in the context of end-user programming languages. The research literature describing code smells mainly focuses on object-oriented languages. Only recently, the research community has started to also include end-user programming

languages. The study of code smells in the context of end-user software development has received growing attention, with notable works studying Yahoo! Pipes web mashups[8] and spreadsheet formulas [9]. Previous experimental results encourage code smell research in the context of end-user software development, suggesting that end-users are aware of code smells and prefer smell-free software [10], [11], [9]

The growing popularity of block-based programming languages as a tool for educational and end-user programming pursuits prompts the research community to take a closer look at the software quality of block-based software. We next describe closely related previous research efforts and explain how our work differs:

a) Identifying potential code smells in block-based projects: We build upon the previous research efforts in identifying code smells in block-based projects. Hermans et al. [12] identify a catalog of 11 code smells in block-based programs written in Kodu and Lego Mindstorms EV3. Our work focuses on the code smells found in Scratch projects, which thus far have not been comprehensively explored. We specifically focus on Scratch due to its enormous success as a tool for introductory computing education. We survey the existing research literature, which serves as the basis of our catalog of 12 Scratch code smells. We define the studied code smells, develop an automated analysis tool, and apply the tool to a large dataset of close to 600K Scratch projects to assess the prevalence and severity of the code smells.

b) Understanding the effects of code smells: Hermans and Aivaloglou [13] conducted a controlled experiment to study how code smells impact novice Scratch programmers. Their results reinforce a shared understanding among computing educators that code smells indeed negatively affect the programmers trying to understand and extend existing code. We derive further empirical evidence of the negative effects of code smells by showing how the high presence of code smells in some popular projects reduces the likelihood of these projects being remixed and extended.

c) Assessing the prevalence of code smells: Prior efforts also focused on assessing the software quality of block-based projects in general and Scratch in particular. Aivaloglou and Hermans[14] analyzed a moderately large dataset of over 250,000 Scratch programs for three code smells: large scripts, dead code, and duplicate block codes. Our work broadens the scope of this prior work on Scratch by considering additional code smells over a larger sample size of Scratch projects. We additionally assess the severity of code smells based on the percentile-based thresholds calculated from the dataset of popular projects, ranked by means of our proposed *Script Addition* heuristic in the top 25%.

IV. Methodology

A. Identifying code smells for Scratch

We derive our catalog of code smells for block-based software by examining the research literature on this topic, considering smells in different types of programming domains. In particular, the code smells we consider in this work are derived from the following categories:

- **Classic code smells:** These language independent code smells have been identified and documented a long time ago, as they universally occur in all types of software domains. The smells in this category include *Duplicated Code* and *Long Script*.
- **Object-oriented code smells:** Scratch supports a limited form of the object-oriented programming style with *scriptable objects*. In a way, each scriptable embodies an object with an encapsulated `private` state that can only be modified in response to receiving external events. In essence, scriptables can be seen as single-instance objects, whose interactions with each other are also subject to the kinds of smells usually found in object-oriented software. In fact, some of the smells in this category have already been identified in the literature. For example, OO-inspired block-based smells, such as *Feature Envy* and *Inappropriate Intimacy* have been identified in reference [12]. The smells in this category include *Middle Man*.
- **End-user code smells:** Some end-user code smells in the literature [8] are also applicable to Scratch. The smells in this category include *Duplicated String* and *Uncommunicative Name*.
- **Block-based code smells:** Some of the smells are unique to block-based software. Although many of these smells share similarities with ones in other categories, certain aspects of Scratch design make these smells unique for this domain. For example, the IDE support for naming sprites is partly responsible for introducing one of these smells, *Uncommunicative Name* [15]. The smells in this category include *No-op*, *Broad Variable Scope*, and *Undefined Block*.

Designating a recurring software quality problem as a smell is a subjective decision. Software domains are known to have unique quality problems [16]. Informed by this insight, we take a conservative approach to introducing new smells, rather preferring to focus on the known smells, identified earlier in each of the categories listed above. In other words, we deliberately disregard potential code smells that are domain-specific, instead focusing on the general smells that commonly occur in Scratch programs. We also disregard those code smells that are unavoidable, possibly due to the limitations of the language. For example, *Duplicated Code* across scriptable objects is not considered because scripts and custom blocks cannot be shared among scriptables. *Long Parameter* is another example deemed irrelevant, as Scratch provides no way to construct data objects as object-oriented languages do to address this problem.

B. Datasets

Scratch currently has over 17.3 million users and over 21 million projects shared[1]. We study Scratch because of its popularity and the accessibility of a large set of projects

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

that this educational / end-user programming community has made publicly available. Our analysis relies on two datasets of Scratch projects, which serve different purposes in our study as described next.

a) A Large dataset of Scratch projects: The first dataset consists of 1,066,308 shared Scratch projects randomly collected during April-July 2016. This dataset will be used in the assessment of the prevalence and severity of code smells in Scratch programs in Section V-B. Out of these projects, we were able to successfully parse and analyze 1,009,192 projects. The disregarded projects were the ones that our parser and analyzer could not handle.

Data exclusion: For the large dataset, the summary statistics in Table I suggests the majority of projects are very small and may not exhibit code smells that we are interested in. To yield meaningful results, we consider projects of sufficient size (20 blocks or more) when assessing the prevalence of code smells.

Based on this criteria, we parsed and filtered the initial project dataset, identifying 594,988 projects deemed as worthy to be analyzed for the presence of smells. In effect, excluding projects on this principle also disregards non-programming projects that contain little interesting programming logic; Dasgupta et al.[17] refer to projects like that as "coloring-contests."

b) Top popular Scratch projects: For the second dataset, we consider 620 popular projects, hosted on the Scratch website[2], as the initial dataset. Out of this dataset, we select the parse-able projects with at least 80 remixes, resulting in 519 projects used for the study. Our reasoning behind this selection procedure is that for a project to get remixed, it has to enjoy popularity.

Table II shows the basic summary statistics of this dataset. As seen, popular projects have a high number of views, marked as "favorite" and "loved" by others, with many remixes. These projects come from a wide range of sizes, based on the number of blocks used (comparable to the lines of code metric).

C. Computing code smell metrics

Next, we describe how the code smell metrics used in this study are computed.

1) Automated smell analysis: We develop an automated code smell analysis tool for Scratch. We leverage a widely used parser generator, Java Compiler Compiler (JavaCC), that takes a grammar as input and automatically generates a parser. As input to JavaCC, we pass the grammar describing the internal representation of Scratch programs. The resulting parser reconstructs the abstract syntax tree (AST) objects from the source code of Scratch projects, already structured using an AST-like representation in the JSON format. We leverage JastAdd [18], a Java-based system for compiler construction that supports the development of analysis tools. Because of the excellent tooling support for compiler-based projects in Java, we used this language to write all our analysis routines.

2) Calculating code smells metrics: There are three possible options for calculating code smells metrics: (1) density per 100 blocks, (2) percentage of smell instance relative to program elements of interest, and (3) instance counts. We make use of all these options, depending on the code smell under study. For example, we use density to adjust for the project size in certain code smells whose incidence tends to increase with the size of the source code. For instance, *Duplicated Code* with the density of 3 means an average of 3 Duplicated Code instances per hundred blocks in the project. We use percentage for those code smells whose incidence tends to grow proportionately to the number of the program elements that can be afflicted by the smell. For example, *Long Script* with 25% means that, on average, 1 out of 4 scripts in the project suffers from the smell. For other code smells with a rare incidence rate expected, we use smell instance counts. We occasionally refer to the three types of smell metrics collectively in the rest of the paper as *code smell incidence rate*. Section V-A defines the studied code smells as well as the options used for calculating them.

3) Code smell analysis parameters: Certain code smells require specifying parameters. Changing these parameter will lead to different results, thus affecting the reproducibility of this work. Although the analysis routines for the majority of analyzed smells require no parameterization, the routines that search for the presence of DC, DS, LS, and II[3] can be parameterized with different sensitivity levels.

For DC, our analysis implementation is based on [19]. The analysis disregards duplicated code segments if they happen to be parts of larger duplicated segments. Our assumption is that the size of a duplicated segment is directly proportional to how harmful the impact of this smell is. Hence, smaller duplicated segments within the larger duplicated segments can be safely ignored without compromising the accuracy of the insights derived from our analysis. This consideration is referred to in the clone detection literature as "clone quality" [19].

Specifically, we consider subtree clones that include nested blocks (e.g., the `while-loop` statement, etc.) and fragment clones (i.e., varying sequences of simple blocks and subtrees), whose minimum size is 8 AST nodes. The analysis considers sequences of up to 10 blocks, which can comprise both simple blocks and subtrees.

For other code smells, we mitigate such subjectivity in choosing the thresholds by relying on a data-driven approach. The two passes of the analysis are required with the first pass extracting the necessary software metrics in the large dataset to determine the appropriate threshold values (e.g., string length and script length across all projects in the large analysis dataset in the case of DS and LS, respectively). We obtain the threshold values similarly to the approach introduced by Lanza and Marinescu [20]. Specifically, we consider the threshold values at the 70th percentile as extreme. Table III presents the thresholds derived from the large analysis dataset.

[2]https://scratch.mit.edu/explore/projects/all/popular (as of March 2017)

[3]Please, refer to Section V-A for the explanation of smell abbreviations.

Statistic	N	Mean	St. Dev	Min	Pctl(25)	Median	Pctl(75)	Max
numBlock	1009192	140.95	571.88	0	9.44	27.93	75.7	25255
numProc	1009192	0.87	7.09	0	0	0	0	947
numScript	1009192	16.76	60.11	0	1.29	3.97	11.06	8020
numSprite	1009192	5.9	9.78	1	1.75	2.95	5.52	595
numVar	1009192	2.78	14.84	0	0	0	0.84	1121
scriptLength	4461713	12.84	24.13	0	4	7	13	1279

TABLE I
BASIC SUMMARY STATISTICS OF LARGE DATASET OF OVER ONE MILLION SCRATCH PROJECTS.

Statistic	N	Mean	St. Dev.	Min	Pctl(25)	Median	Pctl(75)	Max
Views	519	80,317.9	115,952.1	15,780	32,559.5	48,887	83,379	1,916,543
Favorites	519	2,632.9	1,705.8	631	1,579	2,032	3,103.5	13,203
Loves	519	3,180.4	2,006.9	734	1,966.5	2,479	3,775.5	16,419
Remixes	519	593.4	1,347.1	26	137.5	265	505	22,589
Sprites	519	18.9	28.9	0	5	12	23	401
Scripts	519	109.4	246.6	1	23	56	116	3,869
Blocks	519	1,217.2	1,748.0	2	222	621	1,383	14,801

TABLE II
BASIC SUMMARY STATISTICS OF 519 POPULAR SCRATCH PROJECTS

	N	Median	Pctl(70)
scriptLength (LS)	4461713	7	11
foreignAttrAccessNum (FE)	66725	2	4
duplicatedStringGroupSize (DS)	126475	3	4
stringLength (DS)	190881	10	18

TABLE III
THRESHOLDS FOR CERTAIN SMELL DETECTION

D. Script Addition Metric

This metric measures the differences between the original project and its remix. We extract the added code by using a third-party JSON diff tool, which structurally compares the JSON representations of the original project's source code and that of its remixes. The tool produces a comprehensive list of all basic changes (i.e., add, remove, move, replace, and copy), applying which would transform the original JSON file to that of the remix version.

To reliably approximate the actual extent of code changes between the original projects and their remixes, we filter *remove*, *move*, and *copy* to focus on the *add* and *replace* operations. The *replace* operation replaces a script segment with a new one. Although changes made to the JSON source file can be mapped to 4 basic code element types (i.e., scriptable, script, block, literal value), we found changes with a script as the unit of change to be the most meaningful modification. In other words, we do not count block additions to existing scripts. However, we do count all scripts in the new Sprites added to a project. To calculate the *Script Addition* metric, we sum the add and replace operations, and then divide the result by the total number of blocks, so as to adjust for different project sizes. We use the median of Script Addition to represent the average change rate calculated for each pair of original projects and their remixes. For projects with a high number of remixes, we sample 100 remixes randomly to compute this metric.

E. Large-scale assessment of software quality

We conduct a large-scale analysis of over one million Scratch projects leveraging the supercomputing facility at our institution. To process a large volume of computationally intensive program analysis tasks, the infrastructure makes use of the Hadoop MapReduce framework [21]. Our Java-based analysis tool integrates naturally with Hadoop, achieving the required scalability by batch analyzing multiple projects concurrently. The analysis results expressed in the JSON format are stored in the MongoDB database for subsequent data analytics, for which we utilize Apache Spark[22], a cluster computing framework for interactive data analysis. Figure 1 gives a high-level overview of our analysis infrastructure.

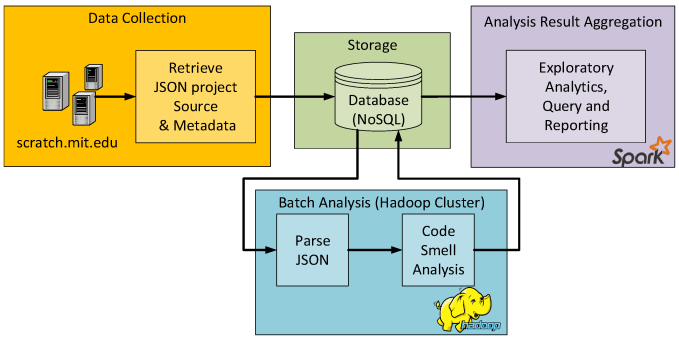

Fig. 1. Large-scale smell analysis infrastructure for block-based software.

V. RESULTS AND TAKEAWAYS

This section presents the results and discusses them in detail for each of the research questions posed.

A. RQ1: Which known code smells can also be commonly found in Scratch programs?

We present a catalog of 12 Scratch code smells. For each identified smell, we also point out its variants in other linguistic contexts. When presenting various cut-off thresholds

for counting a code pattern as a smell, we make use of the statistical thresholds presented and explained in Section V. We derive our statistics-based thresholds from a large dataset of Scratch projects following the approach first presented by [20].

Catalog of code smells in Scratch:

SMELL 1 *Broad Variable Scope (BV)* A variable is marked as broad scope when the variable is made visible to all sprites, but is only used in one sprite. By following commonly accepted design practices, variables should be made local to the scope that uses it. Proper variable scope helps improve comprehensibility as it tells to which sprite the variable belongs. Too many global variables can also be confusing for programmers trying to find the right variable in the script palette and the drop-down menus. Example of smells in other contexts: [23]

SMELL 2 *Duplicate String (DS)*
String values are considered duplicate if the same string values of length at least 18 is used in at least 4 different places. Other contexts: [8]

SMELL 3 *Duplicate Code (DC)*
A fragment of code is duplicated as a way to reuse existing functionality of code at multiple locations in the program. Other contexts: [8]

SMELL 4 *Feature Envy (FE)*
A data producing script is in a different sprite from the one that uses the data. If there is a 1-to-1 relationship between such sprites, they should not be separated, having to talk to each other via a global variable. A better design would have a single sprite, with scripts communicating with each other via a local variable. Other contexts: [6], [12]

SMELL 5 *Inappropriate Intimacy (II)*
A sprite can check on other sprites' attributes through sensor blocks. However, excessively reading of other sprite's private variables (at least 4) can lead to high coupling between sprites. Other contexts: [6], [12]

SMELL 6 *Long Script (LS)*
An unreasonably long script can suggest inadequate decomposition and hinder code readability. A script is considered too long if there are more than 11 blocks measured vertically. Other contexts: [6], [12]

SMELL 7 *Middle Man (MM)*
A long chain of broadcast-receive can be used to pass a message from one script to another. However, using this abstraction to simply delegate work without actions is considered a code smell. Other contexts: [6]

SMELL 8 *No-op (NO)*
A user event-based script that performs nothing can be removed. A common occurrence is event-handling code with no action associated with it. Other contexts: [12]

SMELL 9 *Uncommunicative Name (UN)*
Although other programming entities could suffer a similar effect, this smell focuses particularly on a problematic naming of a sprite —"Sprite" which is highly common name since it is the default name that the IDE gives to generic sprites at the creation time. The evidence of this smell is presented in the work by Moreno and Robles[15]. Other contexts: [6]

SMELL 10 *Undefined Block (UB)*
Scripts can be copied from different projects using the Scratch programming environment feature called "backpack". The scripts with calls to a custom block without its definition will be rendered as undefined blocks, thus ceasing to contribute any useful functionality to the project. This smell is commonly introduced when custom block definitions are not copied and placed first. The rationale of this code smell is similar to *No-op*.

SMELL 11 *Unreachable Code (UC)*
An unreachable script can be safely removed without affecting the program behavior. A script is considered unreachable if it is the receiver of a nonexistent message. This particular case is often caused by removing only the broadcast blocks without adjusting their corresponding receiver blocks. Note that our analysis disregards the fragment scripts not beginning with event blocks, as they are commonly used by Scratch programmers to experiment with code and to initialize persistent data. Other contexts: [12], [8]

SMELL 12 *Unused Variable (UV)*
The Scratch programming environment lets a programmer declare variables in the data palette before they can be used in the scripting area. However, the programming environment provides no support to check if the declared variables are unused and can be safely removed. The rationale of this code smell is similar to that of *Unreachable Code*.

B. RQ2: What can the analysis of popular Scratch projects teach us about the state of software quality in this programming domain?

Scratch projects can be cloned, referred to as "remixed" in the Scratch terminology. The remixes can be traced back to their original projects. In this study, we examine whether popular projects with high code changes in their remixes tend to exhibit higher software quality.

1) RQ2.1: Do projects with a higher Script Addition *metric value exhibit higher software quality?:* We consider popular projects of medium sizes, as defined as being in the range of between the 25^{th} and the 75^{th} percentiles (200-1,400 blocks) of all project sizes. We consider two project subsets that we refer to as:

1) *high-change projects*: 59 popular remixed projects whose *script Addition* metric is ranked in the top 25;
2) *low-change projects*: 87 popular remixed projects whose *script Addition* metric is ranked in the bottom 25;

Table V reports on the medians of the studied code smell metric values for each of these subsets.

We visualize the smell incidence in the two subsets using boxplots, and test if they are drawn from the same distribution

	N	Stdv	Min	Median	Max	Pctl.70.	Pctl.80.	Pctl.90.
BV	51.00	14.60	0.00	15.48	50.00	24.70	32.86	38.57
UC	59.00	0.93	0.00	0.00	5.68	0.00	0.15	0.42
DC	59.00	0.46	0.00	0.44	1.98	0.70	0.87	1.08
DS	25.00	23.09	0.00	0.00	100.00	0.00	0.00	28.57
FE	59.00	5.36	0.00	0.00	27.00	0.00	0.00	6.37
II	59.00	0.25	0.00	0.00	1.00	0.00	0.00	0.00
LS	59.00	10.33	0.00	9.18	40.00	14.40	18.90	29.21
MM	59.00	9.50	0.00	0.83	36.00	6.20	11.20	21.20
NO	59.00	20.38	0.00	2.79	101.00	7.77	19.20	37.70
UN	59.00	34.95	0.00	6.51	100.00	38.25	61.61	89.38
UB	59.00	4.09	0.00	0.00	21.00	0.00	0.73	4.20
UV	51.00	25.24	0.00	14.64	95.24	30.83	44.95	61.25

TABLE IV
SUMMARY STATISTICS OF CODE SMELL OF PROJECTS IN THE BENCHMARKS AND THEIR 70^{th}, 80^{th}, AND 90^{th} PERCENTILE VALUES

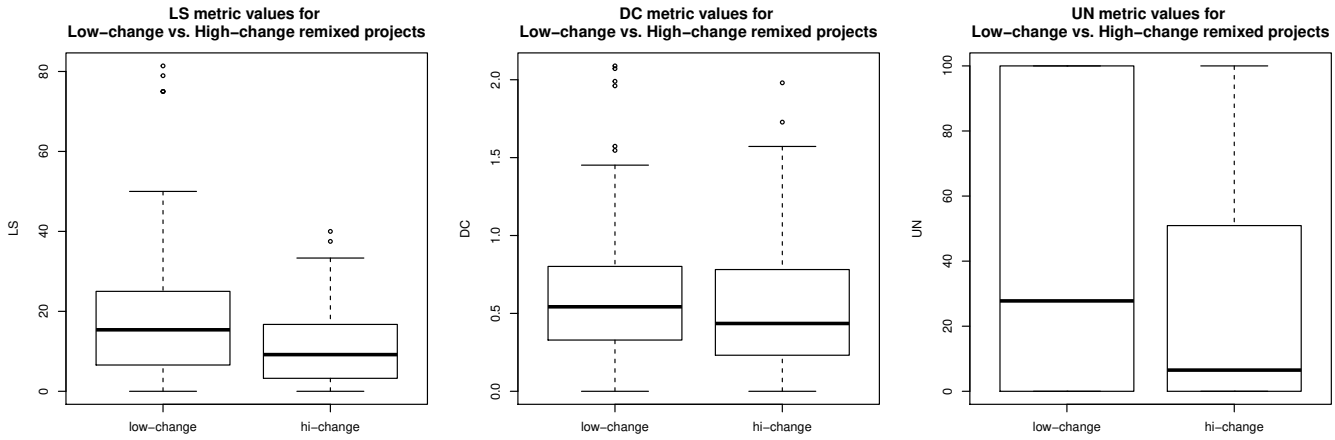

Fig. 2. Incidence rate of code smells LS, DC, and UN in the two subsets (High-change vs. Low-change) of popular Scratch projects

	Low changes (25%)		High changes (Top 25%)		
	Median	SD	Median	SD	P-values
BV	16.67	33.94	15.48	14.54	0.323
UC	0.00	0.99	0.00	0.92	0.997
DC	0.54	0.48	0.44	0.46	**0.044**
DS	0.00	28.62	0.00	22.69	0.202
FE	0.00	4.53	0.00	5.32	0.795
II	0.00	0.21	0.00	0.25	0.707
LS	15.38	17.58	9.18	10.29	**0.003**
MM	0.00	13.64	1.00	9.46	0.980
NO	2.00	18.89	3.00	20.94	0.968
UN	27.78	44.50	6.51	34.84	**0.011**
UB	0.00	3.77	0.00	4.77	0.847
UV	5.56	24.41	15.28	25.02	0.999

TABLE V
THE COMPARISON OF STUDIED CODE SMELL METRICS BETWEEN PROJECTS WITH REMIXED CODE CHANGES IN THE BOTTOM 25% (< 0.1) VS. TOP 25% (> 0.82)

(null hypothesis) using a one-tailed Wilcoxon rank sum test[4]. The null hypothesis is that both of these distributions should be the same. Figure 2 shows boxplots of the LS, DC, and UN smell metrics for the remixed projects, whose *Script Addition* metrics are in the bottom 25% and the remixed projects whose *Script Addition* metrics are in the top 25%. The boxplots show the trends of a lower smell incidence rate in the *high-change* subset. We omitted the boxplots comparing other code smells in the two subsets for brevity, as they do not show any clear trends and their test values are not statistically significant.

The non-parametric Wilcoxon rank sum test (with continuity correction) for LS, DC, and UN conclusively rejects the null hypothesis that the two subsets (low-change remixed projects and high-change remixed projects) are drawn from the same distribution. Table V presents the corresponding effect size (difference of medians) and p-values. As can be observed, the medians of UN and LS smell incidence rates in the high-change subset are dramatically lower than that of the low-change subset. DC has a significant but smaller effect size, based on the median difference of the smell incidence rate.

Takeaway: The low-change projects exhibit software quality that is noticeably worse than their high-change counterparts in the presence of a high incidence rate of *Long Script*, *Uncommunicative Name*, and *Duplicated Code*, known to hinder code comprehensibility and extensibility. Being hard to understand and extend, these projects tend to discourage high-change remixes.

C. RQ3: How prevalent and severe is each studied code smell in Scratch projects?

We study the prevalence of code smells in the dataset of 594,988 projects, selected from the initial 1,009,192 projects

[4]The Wilcoxon rank sum test is a non-parametric test that is not sensitive to outliers, as it does not assume any distribution of sample data.

Smell	N	%Prevalence	%Low risk	%Moderate risk	%High risk	%Very high risk
LS	594,988	51.00	71.13	5.03	8.55	15.30
UN	591,621	48.00	67.36	8.57	5.97	18.10
DC	594,988	35.00	75.86	3.11	3.26	17.77
BV	328,016	30.00	54.86	3.09	3.85	38.20
NO	594,988	28.00	93.71	3.24	1.09	1.96
UV	328,016	20.00	77.51	5.25	6.73	10.50
MM	594,988	17.00	94.02	2.17	1.75	2.06
DS	276,891	13.00	71.71	0.00	14.00	14.29
UC	594,988	9.00	91.30	1.51	1.42	5.77
UB	594,988	8.00	92.29	0.00	5.60	2.11
FE	594,988	5.00	94.63	0.00	2.89	2.48
II	594,988	4.00	95.88	0.00	0.00	4.12

TABLE VI
PREVALENCE AND SEVERITY OF CODE SMELLS IN THE LARGE DATASET

as containing more than 20 blocks as described in IV-B. For each code smell, we count the number of projects as being afflicted by the smell if at least one code smell instance is found. We report on the prevalence of smells as the percentage of the smell afflicted projects over the total number of projects analyzed for the smell of interest. Please note that the number of projects considered for each code smell can vary since certain code smells may not be applicable in some projects and thus we exclude them from the calculation. For example, variable-related code smells (e.g., BV and UV) are not applicable for projects that declare no variables.

Based on our analysis, LS, UN, DC, and BV show high prevalence ($> 30\%$); NO, UV, and MM show moderate prevalence $[10\%, 30\%]$; and DS, UC, UB, FE, and II show low prevalence ($< 10\%$). Additionally, we assess the severity of quality problems by categorizing the projects into increasing risk levels. Code smells with low incidence rate may not be harmful. Hence, one must determine the thresholds exceeding which should classify a project as being at risk. Since choosing threshold values can be subjective, we rely on the approach first introduced by [24] that establishes benchmarks for deriving thresholds. Having shown high *Script Addition* metrics associated with likely high quality projects, we use the high-change subset consisting of 59 projects, described in V-B, as the benchmark for deriving practical threshold values of different smell risks.

The summary statistics of the *high-change* subset as well as the 70, 80 and 90th percentiles, used as the basis for determining the risk intervals are presented in Table IV. We base our intervals on [24] to categorize the severity of quality problems using their percentile values: low (0–70%), moderate risk (70-80%), high risk (80-90%), and very-high risk ($>$ 90%). Table VI presents the prevalence and the severity of the analyzed code smells.

If the quality of an average block-based project is similar to that of popular projects, we expect to see about 10% of the population distributed into each of the moderate, high, and very high risk categories. However, our results show that average projects have been afflicted by code smells differently. That is, BV–38.2%, UN–18.1%, DC–17.8%, LS–15.3%, and DS–14.3% are clearly in the very high risk category. Scratch programmers may be simply unaware or indifferent of these code smells and their harmful effect on program quality. The remaining smells afflict the analyzed projects less commonly. **Takeaway:** For the smells with high prevalence—BV, UN, DC, LS and DS—a larger percentage of average projects is in the very high risk category, which is at odds with the software quality exhibited by the popular projects with high remixability.

VI. THREATS TO VALIDITY

The validity of our analysis results may be threatened by several factors. The documented code smells may not cover all code smells, which are known to be subjective, while additional code smells can appear as new language features and development tools are being introduced. The selection of the benchmark projects may not be appropriate for all types of projects (e.g., long scripts are a common feature of storytelling projects). Our benchmark and its derived thresholds aim at representing a broad category of projects, so as to avoid the manual inspection required to include all types of projects. We establish thresholds by observing the impact on the comprehensibility and extensibility, as guided by our *Script Addition* metric. In other words, these thresholds may not be applicable for studying other aspects of software quality (e.g., reusability, maintainability, etc.). To mitigate the risk of the analyzer producing erroneous results, we manually sample if their subsets adhere to the specified analysis metrics.

VII. CONCLUSIONS AND FUTURE WORK

This work sheds light on the state of quality in block-based software. We provide empirical evidence of quality problems negatively affecting the likelihood of Scratch projects to be remixed and extended. Our large-scale study assesses not only the prevalence of Scratch code smells, but also their severity, presenting conclusive evidence of recurring quality problems in this domain. The results of this work can inform future efforts to support quality improvement practices in block-based programming environments that are aligned with the actual needs of this community.

ACKNOWLEDGEMENTS

This research is supported in part by the National Science Foundation through the Grant DUE-1712131 and the Royal Thai Government scholarship. The authors thank the anonymous reviewers for their thorough comments that helped improve this manuscript.

REFERENCES

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

[2] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some code smells have a significant but small effect on faults," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, p. 33, 2014.

[3] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement*. IEEE Computer Society, 2009, pp. 390–400.

[4] A. Yamashita, "Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data," *Empirical Software Engineering*, vol. 19, no. 4, pp. 1111–1143, 2014.

[5] A. Monroy-Hernández and M. Resnick, "Feature empowering kids to create and share programmable media," *interactions*, vol. 15, no. 2, pp. 50–53, 2008.

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

[7] 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.

[8] 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.

[9] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting and refactoring code smells in spreadsheet formulas," *Empirical Software Engineering*, vol. 20, no. 2, pp. 549–575, 2015.

[10] K. T. Stolee and S. Elbaum, "Identification, impact, and refactoring of smells in pipe-like web mashups," *IEEE Transactions on Software Engineering*, vol. 39, no. 12, pp. 1654–1679, 2013.

[11] C. Chambers and C. Scaffidi, "Smell-driven performance analysis for end-user programmers," in *Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on*. IEEE, 2013, pp. 159–166.

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

[13] 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.

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

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

[16] E. Van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*. IEEE, 2002, pp. 97–106.

[17] S. Dasgupta, W. Hale, A. Monroy-Hernández, and B. M. Hill, "Remixing as a Pathway to Computational Thinking," in *Computer Supported Collaborative Work*. New York, New York, USA: ACM Press, 2016, pp. 1438–1449. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2818048.2819984

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

[19] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 96–105.

[20] M. Lanza and R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[21] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[22] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets."

[23] A. M. Fard and A. Mesbah, "Jsnose: Detecting javascript code smells," in *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*. IEEE, 2013, pp. 116–125.

[24] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–10.