Sniffing Through Millions of Blocks for Bad Smells

Peeratham Techapalokul
Software Innovations Lab, Virginia Tech
tpeera4@cs.vt.edu

ABSTRACT

Code smells codify poor coding patterns known to degrade software quality. Block-based languages have proven to be a viable educational and end-user programming paradigm with increasing adoption across a broad spectrum of users and domains. This rising popularity of this programming paradigm calls for a serious look at the program quality written in block-based languages. While code smells in the context of text-based languages have been studied extensively, the research community lacks a comprehensive understanding of code smells in block-based software.

To address this problem, we present the results of a large-scale study of code smells prevalent in programs written in the highly popular Scratch programming language. We analyzed programs submitted to the public Scratch repository in 2016, considering a million programs altogether. We discovered interesting relationships between the prevalence of certain smells and the levels of proficiency of the programmers commonly introducing them. Our findings not only can help block-based programmers improve the quality of their software, but also establish the requirements for refactoring support in this programming domain.

Keywords

Software quality; Code smells; Block-based programming; Scratch; Introductory computing; CS education

1. PROBLEM AND MOTIVATION

Code smells are patterns indicative of problems in the code, known to degrade program quality. Code smells make programs harder to read, harder to change, and harder to maintain. Code smells is a useful and practical concept in software quality improvement practices. They provide a simple but useful vocabulary for developers to communicate about software quality. For students, being able to recognize code smells is a useful skill to avoid and improve upon bad designs. For developers, code smells guide software quality improvement efforts, such as refactoring.

In this work, we study block-based programming languages, which have been growing in popularity, providing highly effective tools for pedagogical pursuits and end-user, domain-specific development. The resulting increase in block-based software calls for a serious look at its quality. Unfortunately, while smells in text-based program have been thoroughly studied, code smells in block-based software remain poorly understood. This understanding is required to properly inform students and end-users about how to improve software quality by avoiding bad designs. Finally, we plan to follow up on our results by prioritizing our efforts in providing refactoring support for block-based software.

2. BACKGROUND AND RELATED WORK

Previous works identify "bad practices and habits" in Scratch programs, without explicitly identifying them as code smells. Meerbaum-Salant et al.[2] identified scenario-based scripts though intuitive, can lead to poor readability and maintainability. A preliminary study by Moreno[3] uses static analysis to identify 2 bad programming habits (i.e., code repetition and bad object naming) in a 100 Scratch projects.

Aivaloglou and Hermans [1] study over 250,000 projects Scratch programs to understand which types of blocks are used most frequently as well as analyze the subject programs for three code smells: large scripts, dead code, and duplicate block codes. Our work differs by not only considering generic code smells but also block-based specific code smells. We intend to study code smells for block-based languages comprehensively in terms of the number of smells considered (12 smells), and the number of subjects in our study's dataset (∼1M)

3. APPROACH AND UNIQUENESS

Adopting generic code smells as is might not be sufficient or readily applicable to block-based languages due to the unique differences of block-based languages (high-level and domain-specific nature) and the users (the majority of programmers are non-professionals). In this study, we identify code smells commonly found in block-based programming languages in general and Scratch in particular. We also investigate the relationship of projects containing code smells to the programmers' levels of expertise.

Overall, our methodology for identifying a catalog of code smells relies on personal observation, Scratch discussion forums, and those of other researchers and practitioners. We identify a total of 12 distinct code smells with a brief description of each smell as well as its prevalence in Table 1. Our study subjects comprise a collection of 1,066,308

SIGCSE '17, March 8–11, 2017, Seattle, WA, USA.

ACM ISBN 978-1-4503-4698-6/17/03..
DOI: http://dx.doi.org/10.1145/3017680.3022450

Name	Abbrv	Freq (%)	Description
Broad Variable Scope	BVS	56.0	A variable with its scope broader than its usage does not tell which scriptable the variable belongs. Too many global variables clutter script palette and drop-down menus.
Uncommunicative Naming	UN	52.0	Generic naming started with *Sprite* or *message* (e.g. "Sprite2" and "message1") make programs unreadable
Long Script	LS	47.0	Long script (longer than 11 BLOCS) suggest inadequate decomposition and hinder code readability
Duplicate Code	DC	46.0	Repeated sequence of blocks regardless of block arguments is used as a way to reuse code
Unused Custom Block	UCB	29.0	A script definition of an unused custom block can be safely removed without affecting the program behavior
Unused Variable	UV	25.0	A variables is declared but never used anywhere in the program
Unreachable Code	UC	23.0	An unreachable script can be safely removed without affecting the program behavior
Hard-Coded Media Sequence	HCMS	13.0	A sequence of media elements are hard-coded as block arguments
Duplicate String	DS	10.0	Same string values are repeatedly used in multiple program locations
Unorganized Script	US	6.0	Similar event-based scripts are scattered around making the program hard to navigate
Unnecessary Workaround	UW	6.0	Use of polling of flag variables to direct control flow to recreate broadcast-receive mechanism
Extreme Fine-Grained Script	EFGS	2.0	Breaking up of functionally similar scripts into several small fine-grained event-based scripts

Table 1: Description of each smell and the percentage of code smells found in the sample program subjects

Table 2: Distribution of the program subjects

Scratch projects. We define a set of metrics necessary in the study. We develop a code smell analyzer operating at the AST level, and address the scalability challenge by making use of the Hadoop MapReduce on an HPC cluster.

We investigate the relationship between "smelly" programs and their programmers' computational proficiency. We rank programmers by their computational thinking (PCT) scores: 1:basic, 2:developing, 3:proficient. Our PCT score extends the prior computational thinking (CT) metrics [4], which analyzes block-based programs on 7 computational concepts (e.g., data abstraction, flow control, etc.). PCT considers multiple projects written by the same programmer to increase the CT score accuracy.

4. RESULTS AND CONTRIBUTIONS

Table 1 shows how prevalent each smell is, ordered by the most to least prevalent smells, while Table 2 shows the distribution of the subject programs categorized by size and PCT level. The insights we gained from the result of this study can be summarized as follows:

The programming environment is partly responsible for the top two Scratch specific code smells (BVS & UN), particularly, by having global as the default variable scope and auto-generated generic name as default naming for programming elements. The lack of programming support to aid programmers to identify Dead Code smells (UCB, UV, UC) may have caused them moderately prevalent.

Block-based programs are plagued with Duplicate Code, which confirmed the result of previous work[1].

We find interesting insights from the study of relationship between projects containing smells and their project authors' PCT level, shown graphically as a heatmap in Figure 1. First, code smell are rare for programs created by programmers with the PCT=1 regardless of project size level. This indicates the programs these programmers created are not complex enough to exhibit code smells we considered. Overall, a small projects authored by programmers of PCT=2 (*developing* status), are most prone to all code smells. Certain smells (e.g. BVS, UN, and UC smells are less prevalent as projects grow in size as programmers may have been more careful to avoid them to make the program easier to work with). Certain smells are prevalent (US) regardless of the PCT levels of the program authors.

Overall, the findings suggest the need for refactoring tool support for block-based programmers, while the efforts to provide such support should focus on the more prevalent code smells, as trouble spots for the majority of programmers. If block-based programmers can be better informed about which smells to avoid when working with increasingly complex projects, the overall quality of block-based software is bound to improve.[1]

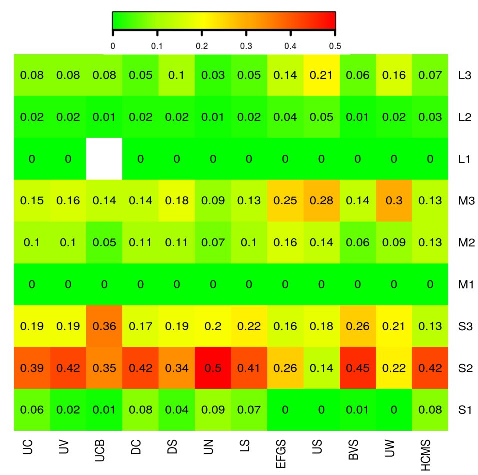

Figure 1: The proportion of code smells across groups of programmers with different PCT levels (1-3) and project sizes(S:small, M:medium, L:large)

5. REFERENCES

[1] 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*, ICER '16, pages 53–61, New York, NY, USA, 2016. ACM.

[2] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari. Habits of programming in Scratch. In *Proceedings of the 16th Innovation and technology in computer science education Conference*, pages 168–172. ACM, 2011.

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

[4] G. Robles, M. Rom, et al. Comparing computational thinking development assessment scores with software complexity metrics. In *2016 IEEE Global Engineering Education Conf. (EDUCON)*, pages 1040–1045. IEEE.

[1]This research is supported in part by the National Science Foundation through the Grant 1134843.