Novice Programmers and Software Quality: Trends and Implications

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

***Abstract*—It remains unclear when it is the right time to introduce software quality into the computing curriculum. Introductory students often cannot afford to also worry about software quality, while advanced students may have been groomed into undisciplined development practices already. To answer these questions satisfactorily, educators need strong quantitative evidence about the pervasiveness of software quality problems in software written by novice programmers. This paper presents a comprehensive study of software quality practices of novice programmers writing Scratch programs. By focusing on finding *code smells*—coding patterns indicative of quality problems—we analyze a longitudinal dataset of 100+ novice Scratch programmers and close to 3K of their programs. Even after gaining proficiency, students continue to introduce the same quality problems into their code, suggesting a need for timely educational interventions. Given the importance of software quality for modern society, computing educators should teach quality concepts and practices alongside the core computing curriculum.**

***Index Terms*—Software Quality; Introductory computing education; Code Smells; Block-based programming; Scratch;**

I. Introduction

To be fully prepared for the challenges of producing high quality software in the real world, students must have been introduced to software quality as part of the curriculum. Nevertheless, the educational community is split on the question of when the right time is to start introducing software quality. Postponing the topic until later in the curriculum often grooms introductory computing learners into undisciplined software development practices. As Will Durant eloquently articulated: "We are what we repeatedly do. Excellence, then, is not an act, but a habit." To truly embrace this principle, we should intrinsically weave software quality into all parts of the CS curriculum, starting from the first programming course.

Prior studies have uncovered the high prevalence of recurring code quality problems in programs written by introductory programmers [1], [2]. This finding calls for a serious reevaluation of the importance of software quality in introductory CS education. However, the research community possesses limited knowledge about the software quality issues of novice programmers. Closing this knowledge gap has potential to provide valuable insights for computing educators, informing the efforts aimed at creating novel educational interventions that integrate the software quality concepts and practices into the CS curriculum.

In this work, we study a large longitudinal dataset of software artifacts produced by introductory computing learners. Our study's goal is to answer the following research questions:

- **RQ1** Does the quality of student programs improve, as students gain programming experience?
- **RQ2**: How persistent are poor coding practices, as students gain programming experience?
- **RQ3** Which programming concepts and patterns lend themselves to influencing the software quality of introductory learners?

To identify quality problems, we adopt the terminology of *code smells*, coding patterns known to indicate possible poor design or implementation choices, the term made popular by Fowler's refactoring book [3]. We analyze a longitudinal dataset of 3,810 Scratch projects, written by a distinct group of 116 novice programmers. For these projects and their programmers, we compute a set of relevant explanatory variables, which comprise the measurements and metrics that potentially associate with the presence of code smells, including programming proficiency and basic programming abstractions and constructs. We apply survival analysis to identify the relationship between a set of explanatory variables and the risk of novice programmers introducing code smells into their programs.

Our results indicate that for *all* levels of programming proficiency, students tend to retain an unchanged attitude toward software quality, irrespective of their current level of programming proficiency. Being exposed to certain programming concepts and constructs lowers the computing learners' vulnerability to introducing some code smells into their projects. However, introducing students to these concepts alone may be insufficient to convince them to embrace software quality. However, if software quality concepts were taught alongside the fundamentals of computing, students would acquire an awareness and practical skills, required to proficiently develop functional computing solutions, while also adhering to well-established software design and implementation practices.

The rest of the paper is structured as follows. Section II discusses the related work. Section III provides background information. Section IV describes our dataset, measurements and metrics used in this work. Section V explains our statistical

2377-570X/17 $31.00 © 2017 IEEE
DOI 10.1109/CSEET.2017.47

Code Smell	Definition
Broad Var. Scope (BV)	A variable with its scope broader than its usage (100% of all variables)
Duplicated Code (DC)	Similar blocks in multiple places ($\geq$ 0.34 instance per 100 LOCs)
Long Script (LS)	A long script with LOCs $>$ 11 (33% instance of all scripts)
Uncommunicative Name (UN)	Poor naming started with "Sprite" (100% of all sprite names)

TABLE I
CODE SMELLS STUDIED IN THIS WORK

analysis and its results. Section VI interprets the results and their implications, and Section VII concludes the paper.

II. Related Work

Recent works offer a strong evidence of quality problems in the context of introductory computing education [1], [4], [2]. However, very few works study software quality practices. In particular, Robles et al. [5] study software clones in Scratch projects. They found no computational concept associated with the absence of duplicated codes, while many students still continue copying and pasting code, despite knowing how to avoid this poor coding practice. The results are similar to the findings of a study of commonly occurring quality issues in student Java codes [6]; students hardly ever resolve quality problems and even the availability of code analysis tools fails to influence how students engage in quality improvement. This work seeks to further investigate the issue and enhance our understanding of persistent quality problems in novice programmers.

III. Background

This section provides the background information required to understand this work.

A. Code smells

Code smells encode those coding patterns that are indicative of possible quality problems. Compared to coding styles, code smells are less subjective, and thus work well in the context of the strict visual syntax of block-based software. Amenable to automated analysis, code smells also allow the analysis heuristics to scale. In this work, we analyze projects authored by novice programmers for the incidence of 4 types of common code smells, which are defined in Table I. Our previous work [2] provides additional information about these code smells.

To account for varying project sizes, we calculate a smell metric based on the percentage or density of smell instances, noted in the Table I (e.g., a density-based metric for *Duplicated Code*, and a percentage-based metric for *Long Script*, etc.). We select the 75th percentile of the smell metric values across all projects in the dataset as the threshold for classifying a project as having an "unacceptable" number of a given smell and being *afflicted* by it (e.g., the threshold for *BV* is 100%). It is these projects that serve as the *events of interest* in our survival analysis.

B. Programming proficiency

To measure programming proficiency of novice programmers, we leverage *Computational Thinking Score (CT Score)*, developed by Moreno and Robles and subsequently evaluated extensively [7]. We calculate six CT dimensions (i.e., abstraction, data representation, flow control, logic, parallelization, and synchronization), disregarding the interactivity dimension, as it is not directly relevant to the general programming proficiency we focus on in this work. The score in each dimension is in the range of 0 to 3, based on the proficiency inferred from the usage of different types of block constructs in the program. The original work by Moreno and Robles [7] provides specific details how these scores are computed.

C. Survival analysis

Survival analysis originated in epidemiology, in which the time to the event-of-interest (survival time) was death, but later has been applied more broadly in different research fields including education (e.g., the effect of remixing or code sharing on student learning progress [8]). Survival analysis addresses the problem of incomplete information about the survival time, called *censoring*. The data may be missing because a study subject has not experienced the event of interest by the time the observation period ends, thus making the information about survival time incomplete. In this study, we apply survival analysis to study the effect of certain learner characteristics (e.g., poor quality of their past projects) on the learners' risk of their project being afflicted by a particular code smell.

A standard way to visually explore and understand survival data is the *Kaplan-Meier* plot of survival against time for each study group; it considers one predictor variable at a time while taking into account the censored data. However, for extensive analysis, we use Cox proportional hazards [9], a popular model for multivariate survival analysis. The Cox's hazard ratio (HR) describes the relative likelihood of the event-of-interest by comparing event rates between different study groups, while adjusting for other significant variables. In this study, the ratios indicate how the *relative likelihood* of the event of interest (the presence of code smells in a project) changes relative to explanatory variables (e.g., numbers of past projects that contain smells).

For example, to study how exposing learners to the concept of procedure affects their risk of introducing duplicate code smell, the following are the possible values of hazard ratio:

HR = 1: at any particular time, the event rates are the same in both groups (the factor has no effect).

HR = 0.5: at any particular time, half as many learners, whose past projects use procedures, are likely to introduce the smell, as compared to the learners, whose past projects use procedures one fewer time.

HR = 2: at any particular time, twice as many learners, whose past projects use procedures, are likely to introduce the smell, as compared to the learners, whose past projects use procedures one fewer time.

Variable Name	Description
prior_{$smell$}	# of $smell$ afflicted projects in the first 10 projects
CT_overall	Median CT scores across all dimensions
CT_{$dimension$}	Mean CT score for a $dimension$
numLocalVar	Median local variables
sensorBlock	Total number of times sensor blocks are used for accessing other sprites' private fields
numProc	Median number of custom blocks (procedures) created

TABLE II
PREDICTORS USED IN THE ANALYSIS MODEL

IV. DATA AND MEASURE

In this section, we describe our approach to data collection, and the set of explanatory variables derived from the data.

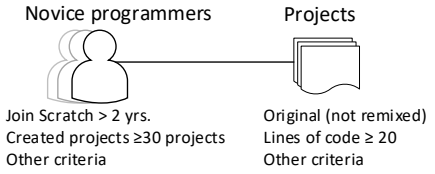

Fig. 1. Criteria for data inclusion

Scratch provides a convenient access to each programmer's shared projects, and their last modification dates. These shared projects form a longitudinal dataset for our case study. Figure 1 provides an overview of the criteria for random selection of the programmers for the study. Each selected programmer must have accomplished a minimum number of projects to ensure sufficient longitudinal data. Project criteria is set to ensure that only sufficiently complex projects are considered, excluding common non-programming projects discussed in [8].

Once the project and programmer data are collected, we then compute the CT score for each dimension. We further exclude non-serious learners, whose entire created projects never reach the overall CT score of 2. To filter out the cases of "an occasional display of proficiency," we only include those projects in which the high proficiency levels (2 and 3) are demonstrated at least three times. We found this heuristic to effectively identify the projects that are worth analyzing in our study. Assuming that programming proficiency is a continuously increasing metric as learners author additional projects, we include in our *analysis dataset* all the subsequent projects once the sought-for level of proficiency is demonstrated. We developed several automated program analysis routines to extract relevant metrics for each project. These metrics represent the explanatory and outcome variables of interest used in our analysis model.

a) Explanatory variables: A set of explanatory variables are used to predict the outcome variable of interest. We use the first 10 projects in the analysis set of each programmer as the *baseline projects*, assuming that the changes in programming proficiency as well as programming practices are small and can be used to capture the programmers' characteristics. Table III gives the summary statistics of the baseline data across all programmers.

b) Outcome variable: The number of projects until the smell afflicted project (exceed the 75^{th} percentile threshold) is used as the time to the event rather than the physical time, a similar approach used in [8]. The smell afflicted projects are identified from the 20 consecutive projects in the analysis dataset, following the initial set of 10 baseline projects. Since programmers may create multiple smell afflicted projects over time, we use a counting process [9], a common approach for modeling the recurring events. This model allows each smell event for the same programmer to be considered independent from each other and contributes to the risk analysis.

Statistic	Mean	St. Dev.	Min	Median	Max
prior_BVS	2.8	2.1	1	2	11
prior_DC	5.2	2.5	1	5	12
prior_LS	5.0	3.1	1	4	15
prior_UN	7.1	3.9	1	7	15
CT_overall	1.4	0.3	0.9	1.4	2.1
CT_parallel	0.9	0.3	0	1	2
CT_dataRep	1.6	0.6	1	1.5	3
CT_abstraction	1.4	0.7	0	1	3
CT_sync	2.2	0.8	0	2	3
CT_flowControl	2.1	0.4	1	2	3
CT_logic	1.6	1.4	0	1	3
numLocalVar	0.2	0.7	0	0	4
sensorBlock	6.2	17.4	0	0	113
numProc	0.2	0.9	0	0	7

TABLE III
BASELINE CHARACTERISTICS OF THE FIRST 10 PROJECTS IN THE ANALYSIS DATASET

V. SURVIVAL ANALYSIS AND RESULTS

In this section, we describe the Cox model used to study the effect of different explanatory variables on the risk of a programmer introducing code smells. We study each code smell using a separate model. The Cox regression models allow adjustment of other explanatory variables in the model, while varying the explanatory variable of interest to ascertain its effect size.

Table IV presents the models and their results. All statistical analysis are performed using R and its survival analysis library (`survival` package). We describe each model, its results, and how the results answer each of the 3 research questions above as follows:

RQ1: proficiency and the risk of poor code quality

To answer RQ1, we include the programming proficiency score as a predictor in each of the models. We control for programming proficiency in each of our analysis models by using the average CT_overall. The results in Table IV show that the CT_overall has no statistically significant effect on the likelihood of novice programmers introducing code smells into their projects. However, gaining programming proficiency in certain CT dimensions can increase the risk of code smells. Specifically, the results show that the increased score for CT_dataRep raises the likelihood of the learner's program to be afflicted by the *BV* smell. *CT_dataRep* has a hazard ratio of 1.74 with statistical significance at 5% level indicated by *. This result suggests 1.74 or 74% more risk relative to the group of learners whose scores are one fewer, when other variables are held constant. Additionally, the increased score

Model	Variable	Hazard Ratio	p-value
BV	prior_BV	1.11	0.005*
	CT_overall	0.48	0.059
	CT_dataRep	1.74	0.001*
	numLocalVar	0.96	0.819
	sensorBlock	0.99	0.045*
DC	prior_DC	1.1	0.000*
	CT_overall	1.08	0.698
	CT_flowControl	1.55	0.000*
	CT_abstraction	0.98	0.768
	numProc	1.02	0.642
LS	prior_LS	1.08	0.000*
	CT_overall	0.12	0.693
	CT_sync	0.78	0.004*
	numProc	0.98	0.725
UN	prior_UN	1.15	0.000*
	CT_overall	0.77	0.264

TABLE IV
THE STATISTICS OF THE EFFECT OF EACH PREDICTOR VARIABLE ON THE LEARNERS' RISK OF INTRODUCING SMELLS.

of *CT_FlowControl* raises the likelihood of the projects being afflicted by *DC* smell.

RQ2: Persistence of poor quality practices

To answer RQ2, we include the prior exposure of a smell, defined as the number of baseline projects being afflicted by the code smell (e.g. *prior_BV*).

Table IV shows how the increased prior exposure to each of the smells has a statistically significant effect on the novice programmers' risk of introducing the smell. For example, prior_BV with HR=1.11 can be interpreted as: for one additional *BV* afflicted project, the programmer has an increased risk of 1.11 times (11%) the risk of those having one fewer *BV* afflicted projects. The Kaplan-Meier curves in Figure 2 visualize varying risks faced by each learner group. Programmers with *BV* afflicted projects in the range (0-3] are more resistant to producing *BV* afflicted smells, as the percentage of such programmers is higher than that in the other two groups, across the observation period.

RQ3: Concepts/constructs for improving software quality

This research question explores how programming concepts/constructs, of which novice programmers may be unaware, can improve their program quality. We take into account some specific Scratch language features. Table V shows code smells and corresponding programming concepts/constructs that are relevant for minimizing each of the smell. These programming concepts/constructs serve as the explanatory variables and are also included in the analysis models.

The results in Table IV show a few programming concepts/constructs appear to naturally induce quality improving practices among novice programmers. Specifically, novice programmers with higher CT_sync have a reduced risk of *LS* afflicted projects (i.e., HR=0.78, or a reduced risk of 22% relative to the group whose CT_sync is one fewer). This finding confirms the observation about scenario-based programming

Code Smell	Scratch Concept/Constructs
BV	Local variable to Sprite, Sensing blocks
DC	Loop iteration, Cloning
LS	Synchronization (broadcast/receive), Procedures
UN	N/A

TABLE V
CODE SMELLS ALONGSIDE PROGRAMMING CONCEPTS/CONSTRUCTS THAT CAN REDUCE THEIR INCIDENCE

[10], which is captured by CT_sync. This programming style fosters modular thinking, which leads to short scripts.

Novice programmers, who have more exposure to the use of sensor blocks in the past, are slightly less likely to introduce the *BV* code smell (i.e., HR=0.99, or a reduced risk of 1% relative to the group less exposed to sensor blocks). Sensor blocks make it possible to read the local variables of other sprites, similar to getter methods to access private fields. However, having been exposed to local variables alone fails to translate into reducing the incidence of *BV* afflicted projects.

Our results show no association between the usage of procedures and the lower risk of *DC* afflicted projects. Upon further investigation, we discovered that very few novice programmers in the dataset have ever used procedures (custom blocks in Scratch).

VI. DISCUSSION

This section discusses our findings and their implications.

a) Informal learning: Our results raise questions about the nature of programming practices fostered by informal programming learning environments, which have become increasingly popular in recent years. In these environments, introductory learners are encouraged to freely explore and learn on their own and from projects shared by others. This way, students move quickly to gain programming proficiency. However, as our results indicate, an increase in programming proficiency does not necessarily translate into proper programming practices, which emphasize the importance of software quality as an important objective.

b) Persistent poor quality practices: Prone to introducing certain code smells into their code, novice programmers continue this trend, as their code continues to be afflicted with the same code smells. These poor quality practices likely need an educational intervention that focuses on how students can avoid introducing them in the first place. Discussing how certain programming practices are considered improper and how to improve upon them can be an effective pedagogical strategy for equipping students with knowledge and skills required to improve software quality.

c) Opportunities for educational intervention: Using certain computing concepts and programming styles can be conducive to decreasing the vulnerability of introducing some code smells, as the protective effect of *CT_sync* on *LS* seems to suggest. This insight suggests that effective educational interventions can introduce known effective programming concepts and practices as a way to improve software quality.

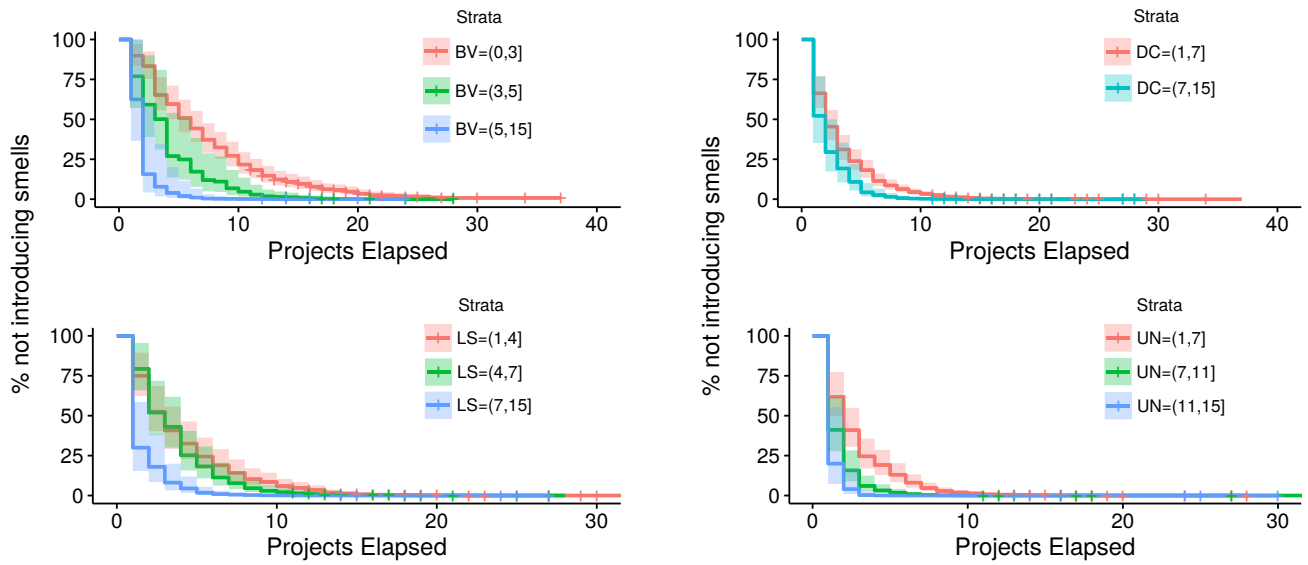

Fig. 2. KM survival curves: Programmers with a high number of prior smell-afflicted projects incur higher risks of introducing the same smells.

d) Threats to validity: Our results might not generalize beyond the context of the study (e.g., the studied subjects beyond the specified criteria, different programming languages, educational settings, etc.). Because of our dataset's properties , the results apply mostly in the context of informal CS education and block-based programming pedagogy. The validity of our findings may be affected by other factors, not considered or impossible to measure in this study, such as age, the setting (i.e., formal / informal settings for CS Education).

VII. Conclusion

Our ultimate objective is to design effective educational interventions to promote the culture of quality and quality improving practices among the introductory computing learners. As a first step in this effort, we conducted this study to understand software quality in the context of novice programmers. Specifically, we strive to understand all the different factors that may affect the software quality of the code written by introductory programmers.

We apply survival analysis to identify the effect of various factors on the programmers' risk of introducing recurring quality problems, known as *code smells*. Our findings show that novice programmers prone to introducing some smells continue to do so even as they gain experience: as their programming proficiency increases, the quality of their code continues to suffer.

These findings indicate the need of promoting the culture of quality from the ground up. To that end, novel educational interventions should be able to instill the importance of software quality in introductory computing learners. By incorporating these insights, novel educational interventions can be designed to seamlessly integrate the core computing concepts with disciplined software development practices, while ensuring that these topics are introduced at the level appropriate for introductory learners.

Acknowledgements

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

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*. ACM, 2016, pp. 53–61.

[2] 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.

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

[4] 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.

[5] 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.

[6] H. Keuning, B. Heeren, and J. Jeuring, "Code quality issues in student programs," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '17. New York, NY, USA: ACM, 2017, pp. 110–115.

[7] J. Moreno-León, M. Román-González, C. Harteveld, and G. Robles, "On the automatic assessment of computational thinking skills: A comparison with human experts," in *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. ACM, 2017, pp. 2788–2795.

[8] S. Dasgupta, W. Hale, A. Monroy-Hernández, and B. M. Hill, "Remixing as a pathway to computational thinking," in *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*. ACM, 2016, pp. 1438–1449.

[9] D. G. Kleinbaum and M. Klein, *Survival analysis: a self-learning text*. Springer Science & Business Media, 2006.

[10] M. Gordon, A. Marron, and O. Meerbaum-Salant, "Spaghetti for the main course?: observations on the naturalness of scenario-based programming," in *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*. ACM, 2012, pp. 198–203.