Understanding Recurring Software Quality Problems of Novice Programmers

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 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 be able to answer these questions, educators need strong quantitative evidence about the persistence of software quality problems in programs written by novice programmers.**

This paper presents a comprehensive study of software quality in programs written by novice programmers. By leveraging the patterns of recurring quality problems, known as code smells, we analyze a longitudinal dataset of more than 100 novice Scratch programmers and close to 3,000 of their programs. Even after gaining proficiency, students continue to introduce certain quality problems into their programs, suggesting the need for educational interventions. Given the importance of software quality for modern society, computing educators should teach quality-promoting practices alongside the core computing concepts.

***Index Terms*—Software Quality; Code Smells; Block-base programming; Introductory CS curriculum**

I. Introduction

Software quality is an essential property for any non-trivial software application. Hence, it is essential that the Computer Science curriculum puts sufficient emphasis on this property, thus fully preparing students for the realities of the real-world software development practices. Nevertheless, the educational community is split on the question of when the issue of software quality should be introduced. The topic of software quality has been traditionally postponed until later in the curriculum, as not being appropriate for the introductory computing learners. Conversely, pushing the topic toward the end of the curriculum may result in negative implications of introductory computing learners being groomed 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." If we are to embrace this principle, then software quality should be intrinsically woven into all parts of the CS curriculum, starting from the first programming course.

In fact, existing studies present strong empirical evidence of the high prevalence of recurring code quality problems [1], [2] in programs written by introductory programmers. These findings help establish the need for a serious reevaluation of how introductory CS education should treat the issue of software quality. However, the research community possesses limited knowledge about the software quality issues as they pertain to novice programmers. Closing this knowledge gap has potential to provide valuable insights for computing educators. These insights can guide the design 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 set of software artifacts produced by introductory computing learners. The goal of our study 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 persistent quality problems, we adopt the terminology of *code smells*, programming patterns known to be indicative of poor designs and/or implementation choices.

In our study, 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, with a particular emphasis on programming proficiency and the use of certain programming abstractions and constructs. We use the Cox regression model—a well-recognized statistical model for survival analysis—to identify the relationship between a

set of explanatory variables and the probability of novice programmers introducing code smells into their programs.

Our results indicate that for *all* levels of programming proficiency, one's attitude toward software quality seems to have a persistent property. In other words, as computing learners are gaining their programming proficiency, the quality metrics of their projects tend to remain constant. The novice programmers who seem careless about the quality of their programs seem to retain this attitude, even as their level of programming proficiency keeps increasing.

Being exposed to certain programming concepts and constructs lowers the computing learners' vulnerability to introducing some code smells into their projects. However, just introducing students to these concepts may be insufficient to improve software quality. In a way, the process of mastering the foundational computing concepts seems unrelated to that of developing an awareness of the importance of software quality. This insight gives rise to a possible educational intervention that teaches computing concepts and programming constructs, while discussing how they can contribute to improving software quality. This intervention would equip introductory learners with an awareness and practical skills, required to proficiently develop functional computing solutions, while also adhering to well-established software design and implementation practices.

The remainder of the paper is structured as follows. Section II provides background on Scratch, code smells, programming proficiency metrics, and the Cox regression model. Section III describes our study's data collection, measurements, and metrics. Section IV describes our statistical analysis model. Sections V and VI present and discuss the results, respectively. Section VII discuss the threats to the validity of our results. VIII discusses related state of the art. SectionIX presents future work directions and conclusions.

II. Background

We begin with an overview of Scratch, a block-based programming language used by the subject programmers of our study, the concept of code smells and software quality metrics and measurements, and finally the Cox model, a statistical model used to study different explanatory variables affecting the risk of programmers introducing quality problems into their programs.

A. Scratch

Scratch is by far the most popular block-based programming languages with over 14.2 million users around the world and more than 17.5 million projects shared [3]. The pedagogical effectiveness and the appeal of block-based programming lie in the block-snapping mechanisms to compose a program, which greatly lower the barrier to entry by eliminating the need to learn language syntax as is the case with text-based programming languages. Scratch programmers use blocks to create a range of interactive and media rich projects, such as games, animation, and storytelling. Programmers can create new projects or remix and modify the existing projects shared by other programmers. Scratch provides a convenient access to the shared projects listed in the chronological order for each programmer. These shared projects form a longitudinal dataset for our case study.

B. Code smells

Code smells are bad coding patterns, indicative of possible design shortcomings, that degrade the software quality of a program. The concept of code smells is often studied in the context of refactoring—behavioral-preserving transformations that improve program quality. The term *code smell* has entered the shared vocabulary of professional software developers after the release of a popular book that described the concept and practices of refactoring [4]. This book documented object-oriented code smells and the corresponding refactoring transformations that can eliminate the smells. Thus, a code smell is a practical software quality concept that provides a simple but valuable vocabulary term that software developers can use to communicate about software quality problems. By identifying the presence of a code smell, a developer indicates the presence of certain quality problems.

In this work, we analyze projects authored by novice programmers for the incidence of four types of common code smells, which are defined in Table I. Because projects authored by such programmers greatly vary in size, it would be meaningless to simply count the total number of occurrences of each code smell under study. Hence, we instead calculate their incidence percentage or density appropriate for each type of code smell. For example, the *Duplicated Code* density is calculated as $cloneCount/100LOC$, while the *Long Script* percentage is calculated as $longScript$ / $totalScript$.

We select the 75$^{\text{th}}$ percentile, of a given smell's density (or percentage) in all the projects in the analysis dataset as the threshold for classifying a project as having a "high" (unacceptable) incidence of code smells. It is these projects that serve as the *events* of interest in the longitudinal dataset in the Cox statistical model, as we describe next.

C. Programming proficiency

To measure programming proficiency of beginner programmers, two proficiency heuristics have been described in the research literature: *programming repertoire* and *Dr. Scratch computational thinking score*. We first briefly define these heuristics and then explain why the Dr. Scratch's metrics is the one that fits the objectives of our study.

Code Smell	Abbrv.	Definition	Threshold	Sources
Broad Variable Scope	BVS	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.	100% of all variables	[5]
Duplicated Code	DC	Repeated sequence of blocks is used as a way to reuse code.	0.34 instance/ 100 LOC	[6]
Long Script	LS	A long script (LOC $>$ 11 BLOCS, derived from the 90th percentile of our analysis dataset) suggest inadequate decomposition and hinder code readability	33% instance/ all scripts	[6], [4]
Uncommunicative Name	UN	Generic naming started with "Sprite" make the program harder to understand	100% of all sprite names	[7], [8]

TABLE I
DESCRIPTION OF CODE SMELLS STUDIED IN THIS WORK AS WELL AS THE THRESHOLDS DERIVED FROM OUR ANALYSIS DATASET

a) Programming repertoire: Brennan and Resnick [9], and later by Dasgupta et al. [10] use the concept of *programming vocabulary* to measure the learning progression. This approach relies on the fact that different types of blocks can be mapped to certain computing concepts. In this approach, programming proficiency is directly correlated with the extent of blocks used (the greater variety of blocks used, the higher the proficiency).

b) Computational Thinking Score (CT Score): The approach is developed and used by Dr. Scratch[1][11], [12], a web-based Scratch project analysis tool. While counting the number of distinct Scratch blocks used in a program can roughly estimate the student's programming proficiency, not all blocks "are created equal" and the mere presence of certain blocks in a program does not always guarantee the actual application of particular programming concepts. The Dr. Scratch CT score aims at addressing this shortcoming: it assigns a block a weight, as based on the respective difficulty of the programming constructs and sometimes the usage patterns it implements. The score is calculated by summing the partial scores assigned to several CT dimensions, as briefly explained in Table II. Each dimension is assigned a score between 0 and 3 points, based on the proficiency level inferred from analyzing the program's code. Note that we disregard the interactivity dimension metric, as it is not directly relevant to the general programming proficiency we study in this work.

Dimension	Description
Abstraction	`whenCloned` $>$ `procDef` $>$ presence of scripts
Data Representation	Usage of list $>$ variables $>$ blocks
Flow Control	`doUntil` $>$ `doRepeat` and `doForever` $>$ presence of scripts
Logic	Logical operation $>$ `IF-ELSE` $>$ `IF`
Parallelization	Advanced event-based block $>$ less advanced ones
Synchronization	`doWaitUntil` $>$ broadcast and receive $>$ wait

TABLE II
DR. SCRATCH'S COMPUTATIONAL THINKING DIMENSIONS. DESCRIPTION LISTS USAGE PATTERNS IN DECREASING ORDER OF PROFICIENCY (FROM 3 TO 1, ORDERED BY $>$).

D. Survival analysis and the Cox model

Originally, survival analysis was applied to those cases, in which the time to the event-of-interest was death. More recently, survival analysis has been applied more broadly, with the events-of-interest including the onsets of diseases, earthquakes, stockmarket crashes, and equipment failure. In the educational context, this statistical technique has been applied to study how the remixing of student programming projects affects the learning progress[10] as well as to investigate factors that may influence students in massive open online courses to drop [13], [14].

In this study, we explore the effect of certain learner characteristics and programming behaviors (e.g., tendency to introduce code smells, usages of certain programming constructs, etc.) on the probability that computing learners will introduce code smells into their projects in the future.

Survival analysis takes into account 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 survival time information incomplete.

A popular statistical model for survival analysis is the *Cox model*, usually written in terms of a *hazard function*, the function of instantaneous hazard of having an event-of-interest at time `t`, given a set of explanatory variables. The hazard ratio (HR) describes the relative likelihood of the event-of-

[1]http://www.drscratch.org/

interest by comparing event rates. 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 a studied predictor (e.g., high and low numbers of past projects that contain smells). For example, in a study of how using procedure constructs affects the duplicate code smell, the hazard ratio can be as follows:

- *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 less frequently.
- *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 less frequently.

III. Data and Measure

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

Figure 1 describes our data collection approach. The first step sequentially checks each project ID for its validity and sharing status, leveraging the fact of Scratch project IDs being auto-generated integers in ascending order. The second step retrieves the author information for all projects whose ID is determined as valid and shared. The following conditions must be met for a user to be included: 1) The author has been a member of the Scratch community for at least two years; this metrics is the difference between the date of sharing the last project and the date of the user joining the community. 2) The author must have created a minimum of 30 projects, excluding the remix projects; this requirement ensures that each included authors comes with sufficient longitudinal data.

Furthermore, non-programming projects, which contain fewer than 20 blocks, are excluded as they may not be sufficiently complex to exhibit code smells and reliably reflect their programmers' proficiency level. This filtering is also applied to prevent common non-programming projects, referred to by Dasgupta et al.[10] as "coloring-contests."

Because Scratch repository only keeps the most recent modified version, all projects shared a long time ago but continuously modified would be excluded. To prevent this mistake, we include all projects, whose last modification date is within 60 days of their sharing date. This provision captures the author characteristics as they move across the different periods of the learning process. These inclusion and exclusion criteria are designed to limit the considered set of authors to those whose projects would yield useful insights for this study.

Variable Name	Description
priorBV	Number of BV smell afflicted projects
priorDC	Number of DC smell afflicted projects
priorLS	Number of LS smell afflicted projects
priorUN	Number of UN smell afflicted projects
numProc	The median number of custom blocks or procedures created
numVar	The median of variables used
numLocalVar	The median of local variables used
cloneBlock	The sum of cloning feature blocks used
CT_(Dimension)	The mean Dr.Scratch CT score for each dimension (i.e. CT_abstraction, CT_dataRep, CT_flowControl, CT_logic, CT_sync)
CT_total	The sum score of each CT dimension
exp	The number of years since the programer joined Scratch at the time of data collection

TABLE III
PREDICTORS USED IN THE ANALYSIS MODEL

Once the project and programmer data are collected, we then compute the CT scores for each dimension as described in Table II. To meaningfully measure program quality, we need to exclude projects authored by complete novices. We define this group as authors whose code never reaches the proficiency level of 2 across all dimensions. 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 techniques to obtain relevant metrics for each project. These metrics represent the explanatory and outcome variables of interests used in our analysis model.

a) Independent variables: Similar to simple linear regression, independent variables or explanatory variables are used to predict the dependent variable, or the outcome variable of interest. As mentioned above, the *Cox model* assumes the time independent property of the explanatory variables. We use the first 15 projects in the analysis set of each programmer as the baseline, with the assumption that in the very beginning, programming proficiency as well as programming habits and practices change little and are thus negligible. We compute the average values for the measurement and the metrics of the first 10 baseline projects to represent the baseline information for each programmer. For most of the measurement and

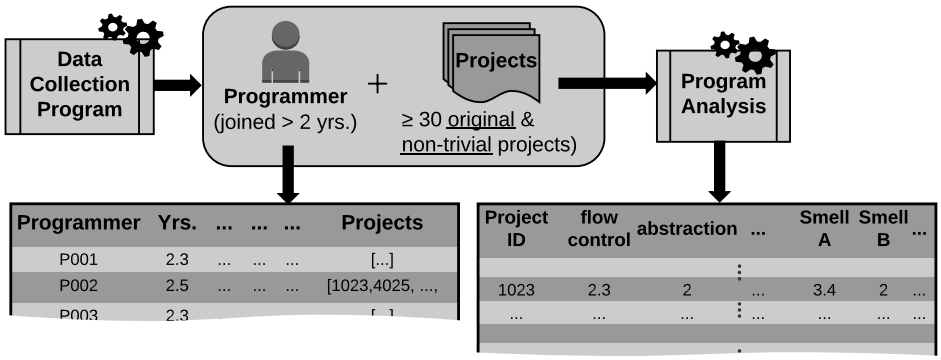

Fig. 1. Overview of the Data Collection Approach

metrics, we calculate the median since the metric data are skewed making arithmetic mean inappropriate. Table IV gives a summary statistics of the baseline data.

b) Outcome variable: Time to event of code smells: Programmers may create multiple projects containing code smells over time. Therefore, code smell events are considered as the recurring events and time-dependent. We describe how recurring events can be transformed to be appropriately used in the model analysis in the next section.

Statistic	Mean	St. Dev.	Min	Median	Max
priorBVS	2.8	2.1	1	2	11
priorDC	5.2	2.5	1	5	12
priorLS	5.0	3.1	1	4	15
priorUN	7.1	3.9	1	7	15
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
CT_average	1.4	0.3	0.9	1.4	2.1
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 IV
BASELINE CHARACTERISTICS OF THE FIRST 15 PROJECTS IN THE ANALYSIS DATASET PRODUCED BY NOVICE PROGRAMMERS

IV. ANALYSIS APPROACH

In this section, we describe how we apply the Cox statistical model to infer how programming proficiency and other relevant explanatory variables affect the risk of a programmer introducing code smells. As explained above, this model is usually expressed in the form of a hazard function. Specifically, this model takes as input "events-of-interest," in this experiment defined as the observed high incidence of code smells (exceed the 75^{th} percentile threshold) in 20 consecutive projects in the analysis dataset, following the initial set of 10 baseline projects which capture the information about their programming practices.

A. Data format for Recurrent Event Modeling

We study each code smell using a separate model for each. All statistical analysis are performed using R [15] and particularly the survival library [16]. Table V shows a simplified input data used to study the impact of various predictors on the programmers introducing *BV* smell. Each row represents a programmer with the baseline explanatory variables and the $etime$'s represent the first five events of code smell afflicted projects. For example, programmer id=2, introduced *BV* code smell in 3 of the first 15 baseline projects. In the observation period of 8 projects (indicated by $futime$), the programmer introduced *BV*-smell afflicted project again in project no.7 (shown in $etime1$ column).

Since projects containing code smell are recurring events, we use the *Counting Process* (CP) format[17], [18] to pass multiple lines of data representing the recurring events for the same individual as input to the Cox model. We transform the time dependent variables of recurring smell events into a new format in Table VI. The new format has two time points ($tstart$, and $tstop$) and the smell status. The programmer id=2 now is represented in two rows, one with time (0-7] with the smell status =1 and the other (7-8] with the status of 0 indicating the censored status (once the subject is censored, no further information is available). Since the same values of explanatory variables apply over the two time intervals, they are the same in the two rows. Note that the analysis time is the project sequence number rather than the physical time, similar to the approach used in [10].

B. Model Development Approach

Two important considerations for model development are the selection of explanatory variables, and the assessment to decide if the model fits well. According to Collett [19], in

id	BVS	views	exp	CC	futime	etime1	etime2	etime3	etime4
1	0	191	2	1	1	NA	NA	NA	NA
2	3	331	2	1	8	7	NA	NA	NA
3	2	309	3	2	5	4	NA	NA	NA
4	0	117	2	1	7	NA	NA	NA	NA

TABLE V
SIMPLIFIED INPUT DATA (THE FIRST FOUR ROWS WITH A REDUCED SET OF EXPLANATORY VARIABLES)

id	BVS	views	exp	CC	futime	tstart	tstop	status
1	0	191	2	1	1	0	1	0
2	3	331	2	1	8	0	7	1
2	3	331	2	1	8	7	8	0
3	2	309	3	2	5	0	4	1

TABLE VI
SIMPLIFIED DATA LAYOUT USED IN THE STUDY (THE FIRST FOUR ROWS WITH A REDUCED SET OF EXPLANATORY VARIABLES)

practice, one selects a set of significant explanatory variables through a combination of knowledge of the science, trial and errors and automatic variable selection procedures. We build separate models for each analysis of code smell by following Collett's approach to model development.

C. Checking proportional hazards assumption

The final model needs to be checked if there exists any interaction between any of the explanatory variables with time. We test the correlation between the *Schoenfeld* residuals and survival time with the null hypothesis being the correlation of zero which supports the proportional hazards assumption[18]. That is, if the test result is highly insignificant, we accept the null hypothesis. We also visually inspect this assumption by plotting a graph of the scaled Schoenfeld residuals, along with a smooth curve that represents (t). If the PH assumption is met, the fitted curve should look horizontal, as the Schoenfeld residuals would be independent of survival time.

D. Using extended Cox model

When the Cox proportional hazard model assumption is violated for a variable, we use the extended Cox model where the variable is stratified. This simply means the effect of such variable is not constant over time. The *stratified Cox model* (SC) model is a modification of the Cox proportional hazard (PH) model to allow for control by "stratification" of the explanatory variable not satisfying the PH assumption. Only variables that satisfy the PH assumption are included in the model as predictors; the stratified variables are not included in the model, though they can still be visualized. The extended model allows the dataset to be divided as strata and create a different baseline hazard for each strata.

E. Interpretation

We use hazard ratio $exp(coef)$ to compare the risk of code smells representing the relative hazard for each one unit increase in the explanatory variable of interest V, holding other variables constant. If the hazard ratio for an explanatory variable is close to 1 then that variable does not affect the risk. If the hazard ratio is less than 1, then the variable is protective (i.e., associated with decreased risk) and if the hazard ratio is greater than 1, then the variable is associated with increased risk. To aid the interpretation of the extent of the effect of different explanatory variables in the model, we plot KaplanMeier estimated survival curves [20], and adjusted survival curves which allow us to visually describe the effect of variables in increasing or decreasing the programmer's risk of introducing code smells in their projects over time. The Kaplan-Meier method [] consider one predictor variable at a time while taking into account the censored data. The adjusted survival curves allow us to visualize the effect of a predictor V, with adjusted survival curves representing hypothetical novice programmers with the adjustments of other significant explanatory variables in the model using their average values, while varying the explanatory variable of interest V.

V. RESULTS

We use separate statistical models to study the effect of a set of predictors on each of the code smells. The results, as presented in Table VII, inform our answers to the research questions first introduced in Section I.

A. RQ1:Does the quality of student programs improve as students gain programming experience?

To answer this research question, 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 overall CT score. The results in Table VII show that the average overall CT has no statistically significant effect on the likelihood of novice programmers introducing code smells into their projects. On the other hand, gaining programming proficiency in certain CT dimensions increases the risk. Specifically, the results show that the increase in the CT data representation score raises the likelihood of learners introducing the *BVS* smell in their programs, while the average CT Flow Control score may raise the likelihood of *DC* smell.

B. RQ2: How persistent are poor coding practices, as students gain programming experience?

To answer this question, we include *Prior Exposure*, a number of times students exhibit poor quality practices, as measured by the presence of a high code smell level in their baseline projects.

Smell	Variables	Multivariate Cox regression	
		Hazard Ratio	p-value
BV	Prior Exposure to the smell	1.11	0.005*
	Avg. variable created	1.01	0.663
	Avg. CT data representation	1.74	0.001*
	Avg. sprite attribute created	0.96	0.819
	Cumulative foreign attribute read	0.99	0.045
	Avg. overall CT score	0.48	0.059
DC	Prior Exposure to the smell	1.1	0.000*
	Avg. procedure created	1.02	0.642
	Avg. overall CT score	1.08	0.698
	Avg. CT flowControl	1.55	0.000*
	Avg. CT abstraction	0.98	0.768
	Cumulative cloning feature count	0.98	0.888
LS	Prior Exposure to the smell	1.08	0.000*
	Synchronization	0.78	0.004*
	Avg. overall CT score	0.12	0.693
	Avg. procedure created	0.98	0.725
UN	Prior Exposure to the smell	1.15	0.000*
	Avg. overall CT score	0.77	0.264

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

The results in Table VII offer a conclusive result that prior exposure to each of the code smells has a statistically significant effect on the novice programmers' risk of introducing the same smells in their future projects.

C. RQ3: Which programming concepts and patterns lend themselves to influencing the software quality of introductory learners?

This research question seeks to explore how different programming concepts and constructs, novice programmers may find themselves unconsciously using, can improve the quality of their programs. To answer this questions satisfactorily, we have to take into account the specific language features found in Scratch. Table VIII presents the mapping between programming practices known to improve code quality and their corresponding sets of programming concepts and constructs available in Scratch.

Note that these programming concepts and constructs serve as the predictor variables and are also included in our analysis models used to answer *RQ1* and *RQ2*.

The results in Table VII show that the programming concepts and constructs appear to naturally induce quality improving practices among novice programmers. Specifically, novice programmers, who have been exposed to sensor blocks, are less likely to introduce the *BVS* code smell. Sensor blocks make it possible to read the local variables of other sprites.

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 VIII
THE MAPPING BETWEEN CODE SMELLS AND THE CORRESPONDING SET OF PROGRAMMING CONCEPTS AND CONSTRUCTS

However, being exposed to the local variables concept alone does not suggest the reduced risk of code smells.

Novice programmers, who have developed the programming style of scenario-based programming, as measured by the *Synchronization* concept, are also less likely to write very long scripts. Scenario-based programming, as suggested by [21], can promote the modular thinking in the resulting codebase, thus improving the overall software quality and reducing the incidence of code smells. Figure 2 shows the effect of each variable on a hypothetical programmer's risk of introducing the *LS* smell over time.

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

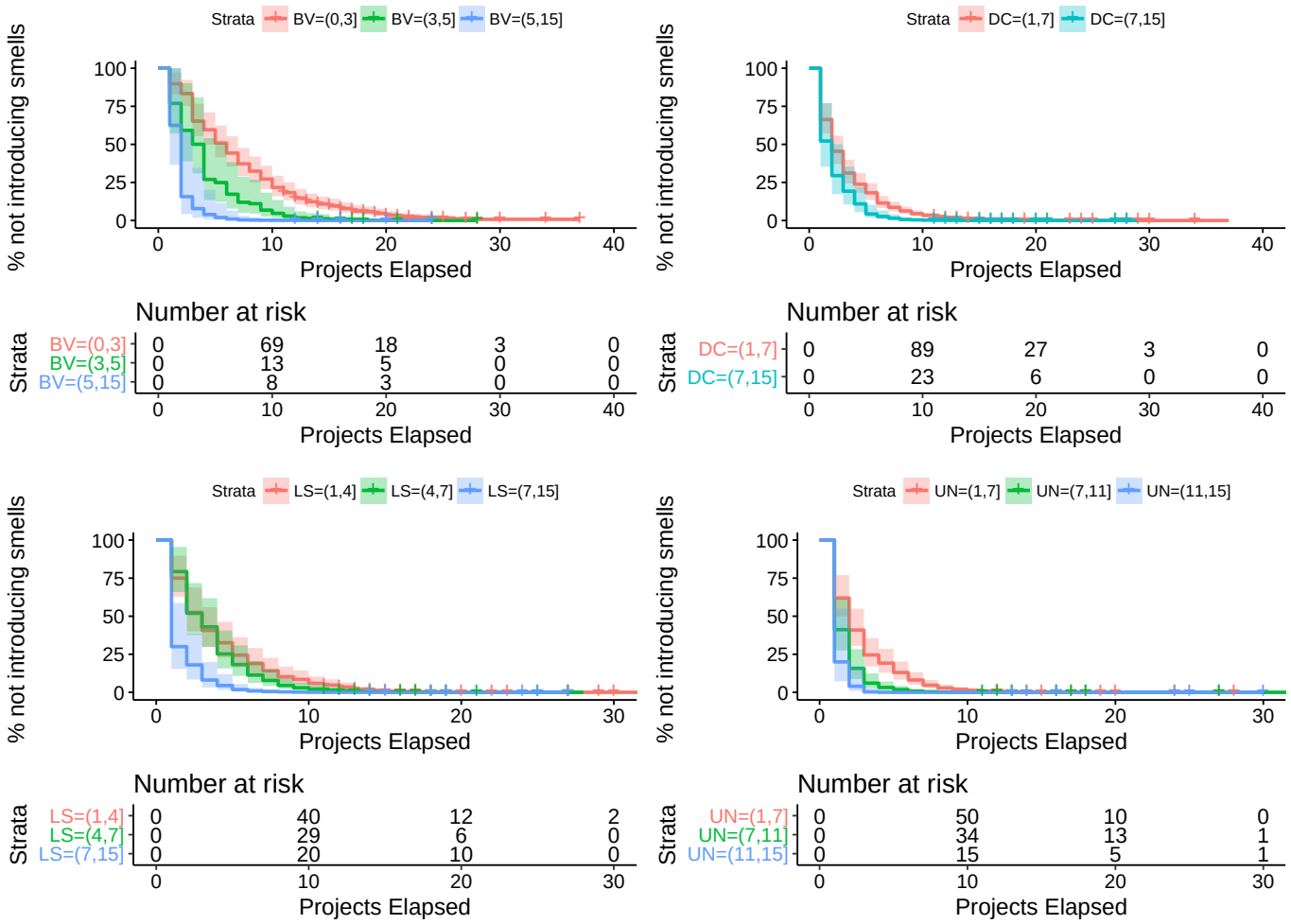

Fig. 2. Kaplan-Meier curves, reflecting the risk of introducing smells by a cohort of novice programmers, as based on their past programming practices of introducing smells

VI. Discussion

Because of the nature of our study dataset, our results may be most applicable in the informal CS education settings. Specifically, 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 experience and proficiency does not necessarily translate into proper programming practices, which emphasize the importance of software quality as an important objective.

Without appropriate educational intervention, poor programming practices can persist and negatively affect the introductory learners. With a sufficiently long time span of (2 years) and a sizeable number of projects, our results provide evidence of persistent quality problems experienced by novice programmers, who start learning how to program at increasingly earlier ages. Undisciplined programming practices, as our analysis results suggest, lead to an increased risk of poorly designed and implemented programs. These results further reinforce the need to educate students about the issues of software quality as part of introductory CS education, in line with suggestions presented in prior works [22].

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.

Novice programmers with a high risk of introducing certain code smells into their programs continue creating software suffering from poor quality; their software continues to be

afflicted with the same code smells, irrespective if the new programming concepts and constructs they have learned and mastered. In other words, certain code smells are not associated with programming proficiency, such as *UN*, and likely need an educational intervention that focuses on how to 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 and follow solid software development practices.

VII. Threats to validity

Our results might not generalize beyond the subjects that we studied. For example, we do not know if the results can be generalizable to a different group of novice programmers and their projects written in different programming languages. To minimize this threat and generalize our findings, one would need to investigate student programs written in different programming languages. We may pursue this investigation as a future work direction. Additionally, many Scratch users are known to learn programming on their own, in unofficial educational settings. Our findings may not apply to students in all educational settings.

Since it is not possible to obtain projects not shared publicly, our study considered only the "shared projects." Our implicit assumption is that unshared projects are work-in-progress, which will eventually be integrated into some version of shared projects appropriate for the study. While analyzing our dataset yields a sufficiently large number of observations, they may not be representative of the entire population. In particular, we consider projects created in the past two years and the programmers who create enough projects to reliably establish the baseline information for each programmer in the analysis dataset. 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).

VIII. Related Work

Block-based programming is quickly gaining steam not only as a popular pedagogical approach to introducing students to computing, but also as a convenient end-user programming tool in multiple domains. Not surprisingly, the research community is becoming increasingly interested in investigating various aspects of block-based programming, with the issue of software quality being at the forefront of several recent efforts described in the literature. In the following discussion, we briefly describe these related efforts and explain how they are related to this work.

Moreno et al. [7] analyzed a dataset of a 100 Scratch projects to study two bad recurring programming habits: sprite/variable naming and duplicated scripts. Their analysis has confirmed that a significant number of the studied projects indeed suffers from these bad programming habits. Aivaloglou and Hermans [1] studied a large-scale sample of Scratch programs (over 250,000 projects) to understand which types of blocks are used most frequently as well as analyzed the subject programs for the presence and prevalence of three recurring code smells: large scripts, dead code, and duplicate block codes. Their findings confirmed that the code smells in question are indeed prevalent in Scratch programs. More recently, Aivaloglou and Hermans Hermans et al. [6] identified 11 code smells in block-based programs written in Kodu and Lego Mindstorms EV3.

A closely related project studies the relationship between code smells and programming experience. Specifically, Hermans and Aivaloglou [23] conducted a controlled experiment to study the effect of code smells on novice Scratch programmers. Their results reinforce a shared understanding among computing educators that code smells indeed negatively affect the programmers trying to understand and modify existing code. Our work builds on these prior efforts by attempting to further identify various factors associated with the risk of the programmer introducing recurring quality problems into their programs.

More recently, Hermans and Aivaloglou [22] created a Scratch-based MOOC course that integrates software engineering principles into an introductory programming curriculum, intended for K-12 students. They report on some promising preliminary results: K-12 students are fully capable of discerning between good and bad software development practices, as well as of avoiding common bad smells that include Code Duplication, Long Script, and Uncommunicative Naming.

A more closely related work by Robles et al.[24] studies software clones in Scratch projects. The authors found no computational concept associated with the absence of duplicated codes, while many students still continue copying and pasting code, despite having the knowledge of how to avoid this harmful coding practice. The results of our work confirm and enhance the findings of these prior efforts that focus on understanding persistent quality problems and their relationship to educational interventions.

IX. 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 what affects software quality. Specifically, in our work, we strive to understand all the different factors that may

influence the software quality for 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, so programming proficiency is not always positively correlated with software quality. These findings indicate the need of promoting the culture of quality from the ground up.

Our findings suggest the need for introducing novel educational interventions that instill the importance of software quality in introductory computing learners. By incorporating these insights, novel educational intervention 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.

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] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari, "Habits of programming in scratch," in *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*. ACM, 2011, pp. 168–172.

[3] Scratch Team. (2017) Scratch statistics. [Online]. Available: https://scratch.mit.edu/statistics/

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

[5] 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.

[6] 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.

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

[8] 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

[9] K. Brennan and M. Resnick, "New frameworks for studying and assessing the development of computational thinking."

[10] 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

[11] 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.

[12] J. Moreno-Len, G. Robles, and M. Romn-Gonzlez, "Comparing computational thinking development assessment scores with software complexity metrics," in *2016 IEEE Global Engineering Education Conference (EDUCON)*, April 2016, pp. 1040–1045.

[13] D. Yang, T. Sinha, D. Adamson, and C. P. Rosé, "Turn on, tune in, drop out: Anticipating student dropouts in massive open online courses," in *Proceedings of the 2013 NIPS Data-driven education workshop*, vol. 11, 2013, p. 14.

[14] D. Yang, M. Wen, I. Howley, R. Kraut, and C. Rose, "Exploring the effect of confusion in discussion forums of massive open online courses," in *Proceedings of the Second (2015) ACM Conference on Learning@ Scale*. ACM, 2015, pp. 121–130.

[15] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2016. [Online]. Available: https://www.r-project.org/

[16] T. M. Therneau and T. Lumley, "Package survival," 2016.

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

[18] T. Therneau, C. Crowson, and E. Atkinson, "Using time dependent covariates and time dependent coefficients in the cox model," 2017.

[19] D. Collett, *Modelling survival data in medical research*. CRC press, 2015.

[20] J. T. Rich, J. G. Neely, R. C. Paniello, C. C. Voelker, B. Nussenbaum, and E. W. Wang, "A practical guide to understanding Kaplan-Meier curves," *Otolaryngology-Head and Neck Surgery*, vol. 143, no. 3, pp. 331–336, 2010.

[21] 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.

[22] F. Hermans and E. Aivaloglou, "Teaching software engineering principles to K-12 students: A MOOC on Scratch," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering and Education Track*, ser. ICSE-SEET '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 13–22. [Online]. Available: https://doi.org/10.1109/ICSE-SEET.2017.13

[23] ——, "Do code smells hamper novice programming? a controlled experiment on scratch programs," in *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE, 2016, pp. 1–10.

[24] 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.