Do Code Smells Hamper Novice Programming?

A Controlled Experiment on Scratch Programs

Felienne Hermans, Efthimia Aivaloglou
{f.f.j.hermans, e.aivaloglou}@tudelft.nl
Delft University of Technology
The Netherlands

***Abstract*—Recently, block-based programming languages like Alice, Scratch and Blockly have become popular tools for programming education. There is substantial research showing that block-based languages are suitable for early programming education. But can block-based programs be *smelly* too? And does that matter to learners? In this paper we explore the *code smells* metaphor in the context of block-based programming language Scratch. We conduct a controlled experiment with 61 novice Scratch programmers, in which we divided the novices into three groups. One third receive a non-smelly program, while the other groups receive a program suffering from the *Duplication* or the *Long Method* smell respectively. All subjects then perform the same comprehension tasks on their program, after which we measure their time and correctness. The results of the experiment show that code smell indeed influence performance: subjects working on the program exhibiting code smells perform significantly worse, but the smells did not affect the time subjects needed. Investigating different types of tasks in more detail, we find that *Long Method* mainly decreases system understanding, while *Duplication* decreases the ease with which subjects modify Scratch programs.**

I. Introduction

Scratch is a programming language developed to teach children programming by enabling them to create games and interactive animations. The public repository of Scratch programs contains over 12 million projects. Scratch is a *block-based* language: users manipulate blocks to program. Block-based languages are visual languages, but also use some successful aspects of text-based languages such as limited text-entry and indentation, and as such are closer to 'real', textual programming than other forms of visual programming, like dataflow languages are.

Block-based languages have existed since the eighties, but have recently found adoption as tools for programming education. In addition to Scratch, also Alice [1], Blockly[1] and App Inventor [2] are block-languages aimed at novice programmers.

In this paper we explore code smells within the context of block-based languages. Code smells, a concept originally coined by Martin Fowler [3], were designed to indicate weak spots in object-oriented source code that are in need of improvement, i.e. refactoring. Well-known examples of code smells are long methods or duplication in source code.

Because code smells were designed for object-oriented source code, this raises the question whether they can also occur in other programming paradigms, and whether they should be considered harmful there too. Recently, it has been demonstrated that code smells occur in various alternative programming environments including spreadsheets [4], [5], Yahoo! Pipes [6] and LabView [7].

The above studies showed that end-users recognize smells [4], [5] and prefer the non-smelly version of programs [6]. For example, the latter found that 63% of Yahoo! Pipes users preferred non-smelly pipes. The preference for clean pipes increased to 71% for tasks involving laziness and redundancy smells. Furthermore, Yahoo! Pipes users performed better on non-smelly pipes: Stolee and Elbaum furthermore show that smelly pipes lead to worse performance when asking users to identify a pipe's output. Given a clean pipe users succeeded 80% of the time, while for smelly pipes this number dropped to 67%. Subjects' completion times also increased: analysis of smelly pipes took on average 68% longer than for clean pipes.

Is this paper we explore whether we observe similar patterns when looking at novices programming with Scratch. As such, the goal of this paper is to **investigate whether code smells impact novices' comprehension of Scratch code**. To address this goal, we conduct a controlled experiment using Scratch with 61 high-school children. We create three different versions of a Scratch program implementing the 'Pong' game. The first version of this program is a 'non-smelly' version, in a second version we introduced the *Long Method* smell, and the final version suffered from the *Duplication* smell. We then divide the subjects into three random groups, and have them perform the same comprehension tasks on their program, including explaining the program's behavior and modifying it. We measure both the time to completion and the correctness of the tasks of the three groups. The results reveal that the novice Scratch programmers subjects perform significantly better on the smell-free programs, but that there is no difference in their completion time. Investigating different types of tasks in more detail, we find that *Long Method* mainly decreases subject's understanding of the game as a whole, while *Duplication* makes it harder for subjects to modify their Scratch programs.

The contributions of this paper are as follows:

- The definition of code smells in the context of Scratch programs (Section IV)
- An empirical evaluation of the impact of these smells (Section VI)

[1]https://developers.google.com/blockly/

II. Background and Motivation

Block-based languages go back to 1986, when Glinert introduced the BLOX language [8]. BLOX consists of puzzle-like programming statements that can be combined into programs by combining them both vertically and horizontally. After a decade of little activity into block-based languages, they became a research topic again, starting with Alice [1]. More recently, new block-based languages have gained widespread popularity, especially powered by Scratch [9] and Blockly[2]. Over 100 million students have tried Blockly via Code.org, and the Scratch repository currently hosts over 12 million projects. Unlike in BLOX, in these new block-based languages the programming blocks can only be combined vertically, resembling textual code more.

Since their introduction, studies have demonstrated the applicability of block-based languages as a tool for education. Scratch, for example, was evaluated with a two-hour introductory programming curriculum for 46 subjects aged 14 [10]. This study indicated that Scratch could be used to teach computer science concepts: analysis of the pre- and post-tests showed a significant improvement after the Scratch course, although some concepts like variables and concurrency remained hard for students.

Moskal *et al.* [11] compared computer science students who studied Alice before or during their first programming course to students that only took the introductory computer science course. Their results show that exposure to Alice significantly improved students' grades in the course, and their retention in computer science in general over a two year period. A follow-up study by Cooper *et al.* [12] obtained similar results, showing that a curriculum in Alice resulted in improved grades and higher retention in computer science.

Most convincingly, Price and Barnes performed a controlled experiment in which students were randomly assigned to either a text-based or a block-based interface in which they had to perform small programming tasks [13]. Their experiment showed that students in the block-based interface were more focused and completed more of the activity's goals in less time.

Summarizing the above, we conclude that block-based languages have a clear potential to be a great tool for introductory programming education, in some cases even outperforming text-based languages.

III. Relevant Scratch Concepts

This paper is by no means an introduction into Scratch programming, we refer the reader to [14] for an extensive overview. To make this paper self-contained, however, we explain a number of relevant concepts in this section.

Scratch is a block-based programming language aimed at children, developed by MIT. Scratch can be used to create games and interactive animations, and is available both as a stand-alone application and as a web application. Figure 1 shows the Scratch user interface in the Chrome browser.

[2]https://developers.google.com/blockly/

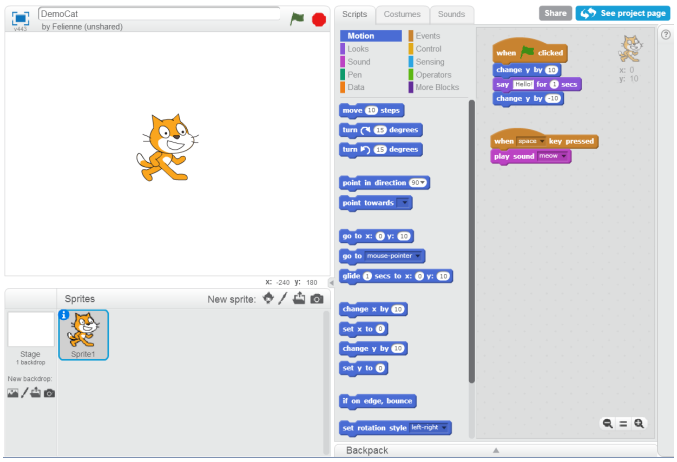

Fig. 1. The Scratch user interface consisting of the 'cat' sprite on the left, the toolbox with available blocks in the category 'motion' in the middle and the code associated with the sprite on the right. The upper right corner shows the actual location of the sprite.

A. Sprites

Scratch code is organized by 'sprites': two-dimensional pictures each having their own associated code. Scratch allows users to bring their sprites to life in various ways, for example by moving them in the plane, having them say or think words or sentences via text balloons, but also by having them make sounds, grow, shrink and switch costumes. The Scratch program in Figure 1[3] consists of one sprite, the cat, which is Scratch's default sprite and logo. The code in the sprite will cause the cat to jump up, say "hello", and come back down, when the green flag is clicked, and to make the 'meow' sound when the space bar is pressed.

B. Events

Scratch is *event-driven*: all motions, sounds and changes in the looks of sprites are initiated by events. The canonical event is the 'when Green Flag clicked', activated by clicking the green flag at the top of the user interface. In addition to the green flag, there are a number of other events possible, including key presses, mouse clicks and input from a computer's microphone or webcam. In the Scratch code in Figure 1 there are two events: 'when Green Flag clicked' and 'when space key pressed'

C. Scripts

Source code within sprites is organized in scripts: a script always starts with an event, followed by a number of blocks. The Scratch code in Figure 1 has two distinct scripts, one started by clicking on the green flag and one by pressing the space bar. It is possible for a single sprite to have multiple scripts initiated by the same event. In that case, all scripts will be executed simultaneously. For example, the code on the left of Figure 5 has five scripts associated with the 'when Green Flag clicked' event.

[3]https://scratch.mit.edu/projects/97086781/

D. Remixing

Scratch programs can be shared by their creators in the global Scratch repository[4]. Shared Scratch programs can be 'remixed' by other Scratch users, which means that a copy of this program is placed in the user's own project collection, and can be then further changed. The 'remix tree' of projects is public, so users can track which users remix their programs, a bit similar forking in GitHub. Contrary to forking though, changes upstream cannot be integrated back into the original project.

IV. Definition of smells

We follow the approach of earlier work in smell detection for end-user languages, where a mapping is defined between OO concepts and concepts in the non-traditional programming language. In spreadsheets, for example, worksheets are mapped to classes and formulas to methods [5]. In work on Yahoo! Pipes, modules are mapped to classes and subpipes to methods [6]. In Scratch, we analogize a script (as explained above, this is a group of blocks started by an event) to a method, as scripts represents one unit of computation. As such, an event that has a large number of associated blocks can be considered to suffer from the *Long Method* smell. The second smell that we investigate is the *Duplication* smell, in which the similar code is repeated in multiple parts of the program. This inhibits easy maintenance, as developers have to make similar changes within the different 'clones'. Summarizing, we study the following code smells:

Long Method Scratch code suffers from the *Long Method* smell if a group of blocks grows very large. In that case, it will be hard to understand and modify.

Duplication Scratch code suffers from the *Duplication* smell when similar computations or events occur in multiple places in the program, either within or across sprites.

V. Research Questions and Experimental Setup

The goal of this paper is to investigate the effect of code smells on the performance of Scratch users. Similar to other experiments on program comprehension [15], [16] we make a distinction between time spent on the tasks and their correctness. As such, we focus on the following three research questions:

1) Does the presence of the code smells *Long Method* or *Duplication* in Scratch increase the time that is needed to complete typical comprehension tasks?
2) Does the presence of the code smells *Long Method* or *Duplication* in Scratch decrease the correctness of the solutions given during those tasks?
3) Are there certain types of comprehension tasks are disproportionately affected by the code smells *Long Method* or *Duplication*?

Associated with the first two research questions are two null hypotheses, which we formulate as follows:

[4]https://scratch.mit.edu/explore/projects/all/

$H1_0$ The presence of the code smells *Long Method* or *Duplication* does not impact the time needed to complete typical comprehension tasks.

$H2_0$ The presence of the code smells *Long Method* or *Duplication* does not impact the correctness of the solutions given during comprehension tasks.

The alternative hypotheses that we use in the experiment are the following:

$H1_1$ The presence of the code smells *Long Method* or *Duplication* increases the time needed to complete typical comprehension tasks.

$H2_1$ The presence of the code smells *Long Method* or *Duplication* decreases the correctness of the solutions given during comprehension tasks.

To test the null hypotheses, we create three versions of a simple Scratch game: a non-smelly version and two smelly versions: one exhibiting the *Long Method* smell and one with the *Duplication* smell. For these three versions of a Scratch game, we define a series of typical code comprehension tasks that are both to be solved by a three groups, each working on one of the three versions. We use a between-subjects design, meaning every Scratch user is either in the control group or in one of the two experimental, 'smelly' groups.

A. Subjects

We perform our experiment with 61 high-school children, in the first year of Dutch high-school. Note that Dutch high-schools start at the age 12 rather than at the age of 14 as common in many other countries. Before the experiment we gathered personal information from the subjects. The subjects vary in age between 12 and 14, with an average age of 12.8. Out of the 61 subjects, 45 had no prior exposure to programming, while 16 did have programming experience, either in Scratch or in the LEGO Mindstorms environment. We divided the experienced children equally over the different test groups, as shown in Figure 2. Among the subjects were 38 boys and 23 girls. We did not distribute them evenly over the test groups as gender was not one of the variables under study. However, the genders turned out to be balanced quite evenly, as demonstrated by Figure 3.

B. Program under study

As program in our experiment, we selected a project from the Creative Computing Handbook [14]. This book consists of numerous animation and game ideas, plus the corresponding Scratch code. While called Pong, this game is not the classical Pong where two players have to hit a ball. The aim of this Pong game is to ensure that the ball does not touch the lower line (red in Figure 4). In that case, players lose all their points. For every time the player hits the ball, they earn a point.

C. Smells

We implemented three different versions of the Pong program: one without the smells, one exhibiting the *Duplication* smell and one suffering from the *Long Method* smell.

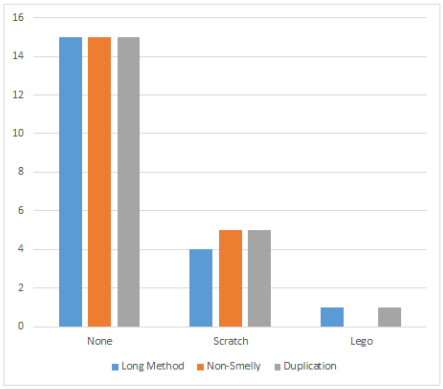

Fig. 2. Subject's previous exposure to programming divided over the three different test groups

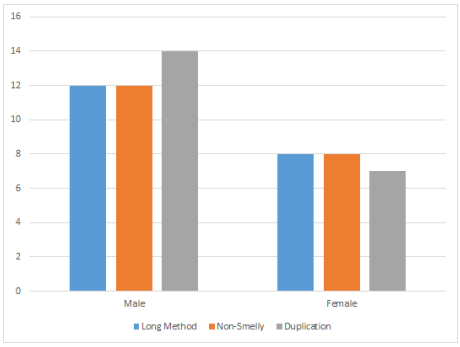

Fig. 3. Subjects' gender divided over the three different test groups

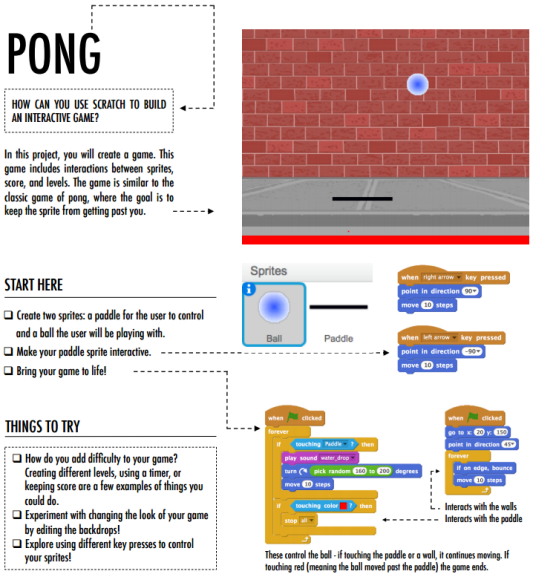

Fig. 4. The game of Pong programming instructions as taken from the Creative Computing Handbook [14].

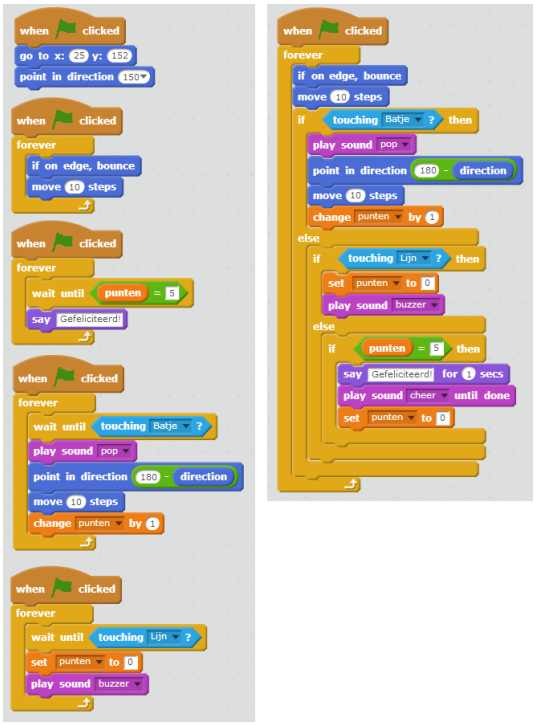

Fig. 5. Two versions of the movement logic for the Ball sprite in Pong, on the left hand side the non-smelly version and on the right hand side the version suffering from a *Long Method* smell.

The three versions are available in the Scratch repository[5]. We have introduced smells in the programs as follows:

1) Long Method *: Long Method* is introduced in the Ball sprite, by combining as much logic of the sprite into one script. For example, instead of having one block for movement and one to detect touching, everything is combined together within an if statement. Figure 5 depicts two versions of the logic for the movement of the Ball sprite in Pong. The non-smelly version on the left has five separate scripts, one for the start position, the movement, an event in case of 5 points, and touch detection for the Paddle and Line respectively, while the smelly one combines everything in one script.

2) Duplication *: Duplication* is introduced in the program by having duplicated events in different sprites. Figure 6 shows two versions of our programs, the non-smelly version on the left and the version suffering from *Duplication* on the right. The figure shows that, for example, both the Ball detects a touch of the Paddle, and the Paddle detects a touch of the Ball. The Ball sprite controls the movement and the points while the Paddle creates the sound. In the non-smelly version on the left there is just one script containing all actions when the Ball touches the Paddle.

D. Tasks

To select appropriate comprehension tasks, we employ the comprehension framework from Pacione *et al.* [17]. Pacione

[5]https://scratch.mit.edu/users/Felienne/

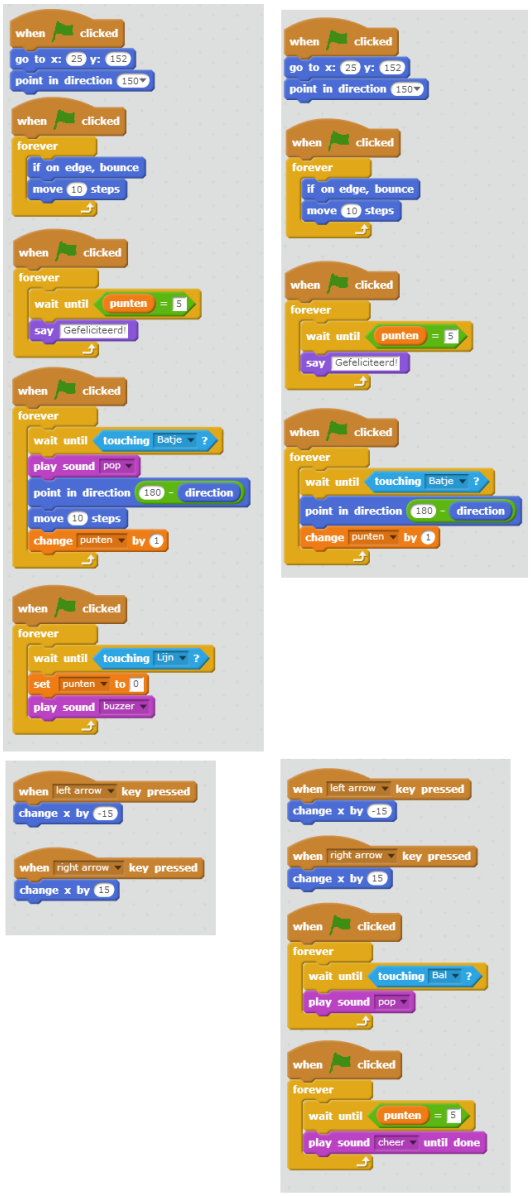

Fig. 6. Two versions of the movement logic for the Ball (top) and Paddle (bottom) sprites in Pong, on the left hand side the non-smelly version and on the right hand side the version suffering from the *Duplication* smell, since the 'wait until 5 points' block occurs in two sprites.

studied several sets of tasks used in software comprehension literature and divided them into nine distinct activities. Table I presents an overview of Pacione's categories. Table II lists the tasks we use in our experiment mapped to Pacione's tasks. We divide the tasks into three different categories. Tasks T11 to T12 are tasks in which subjects have to explain the code of one single sprite. In tasks T21 to T23 subjects have to explain the functionality of the entire system, and in tasks T31 to T33 finally, subject have to make modifications to the game's functionality.

We print all tasks on paper answer sheets on which the children have to write their answers. We chose a paper answer

TABLE I
PACIONE'S COMPREHENSION ACTIVITIES

Activity	Description
A1	Investigating the functionality of the system
A2	Adding to or changing the system's functionality
A3	Investigating the internal structure of an artifact
A4	Investigating dependencies between artifacts
A5	Investigating run-time interactions in the system
A6	Investigating how much an artifact is used
A7	Investigating patterns in the system's execution
A8	Assessing the quality of the system's design
A9	Understanding the domain of the system

sheet rather than a digital one because we felt that switching between Scratch and a digital answer sheet on a computer, in a computer room with relatively small screens, would make the experiment unnecessarily difficult for participants.

The first and second group of tasks, code explanation (T11 and T12) and system explanation (T21 to T23) are performed by subjects by only looking at the Scratch code on the paper answer sheets, on which we included screenshots of the Scratch code for the two different sprites: the Ball and the Paddle. Figure 7 shows such a page from the workbook.

When subjects start the modification tasks, they are allowed to open the program on their computers. For that, the workbook contains a link to one of the three versions of the Pong game, which subjects were then asked to 'remix'. For the modification tasks (T31 to T33) the subjects both have to write down a strategy for how to address the change on their paper workbooks, as well as perform the described change in Scratch and save their program.

Note that Table II does not have a task associated with A8 "Assessing the quality of the system's design". We do however close the experiment by asking subjects questions that reflect on the experiment: including whether they like the program and why, and whether they like programming in general and why. In that sense, we have included a task for A8. However we do not count the scores for that task towards subjects' performance and therefore it is not present in the table. We exclude this task because we feel that, contrary to professional developers for which Pacione made the framework of activities, judging a system's design is not a core competency for novices. Furthermore, we did not want to prepare the children for assessing quality, by explaining them what type of code is good or bad, as that might influence their performance.

E. Experimental procedure

The experiment is conducted during three guest lectures by the authors at a high school in Delft. Every session lasted two

TABLE II
COMPREHENSION TASKS USED IN THE SCRATCH EXPERIMENT AND THEIR MAPPING TO PACIONE'S COMPREHENSION ACTIVITIES (TABLE I). WE DISTINGUISH THREE DIFFERENT GROUPS OF TASKS: CODE EXPLANATION (T11 AND T12), SYSTEM UNDERSTANDING (T21 TO T23) AND SYSTEM MODIFICATION (T31 TO T33)

Task	Description	Points	A1	A2	A3	A4	A5	A6	A7	A8	A9
T11	What does the code in the ball sprite do?	1	X		X						
T12	What does the code in the paddle sprite do?	1	X		X						
T21	What are the ways to get points in the pong game?	1	X			X	X	X	X		X
T22	When have you won the game and what happens then?	2	X			X	X		X		X
T23	When have you lost the game and what happens then?	2	X			X	X		X		X
T31	Change the game so that is runs up to 10 points.	2	X	X				X			
T32	Make the ball say something if you lose.	2	X	X				X			
T33	Make the ball say something if the ball hits the paddle.	2	X	X				X			
Total		13									

Fig. 7. A page from the paper workbook on which subjects filled out their answers (in Dutch).

hours, of which the first lesson was spent teaching the children to program in Scratch, by making the maze game using a step-by-step tutorial. This tutorial is available online[6], but note that it is in Dutch. After the tutorial hour, the experiment starts. The subjects are then given one of the three versions of the Pong game and all have to answer the same tasks, as summarized in Section V-D.

F. Variables and Analysis Procedure

The independent variable in our experiment is the presence of code smells in the programs under study. The first dependent variable is the time spent on the comprehension tasks as listed in Table II. Time needed is measured by having the subjects write down the current time on their answer sheets every time they start a new task. Figure 7 shows a box for the time on the right. The second dependent variable is the correctness of the given solutions. This is measured by applying our solution model to the subjects' solutions, which specifies the required elements and the associated scores, further detailed in Section VI-B.

[6]http://www.felienne.com/archives/4587

VI. RESULTS

A. Time Needed to Complete the Exercises

Figure 8 shows an overview of the completion times of all tasks (T11 to T33). A Shapiro-Wilk test showed that all three samples follow the normal distribution, with means of respectively 21.6, 20.0 and 21.9 and standard deviations of 5.2, 5.2 and 6.3. Furthermore the three groups have equal variance as demonstrated by Levene's test. An ANOVA test however resulted in a p-value of 0.520, meaning we cannot reject $H1_0$. In other words there is no significant difference between the three test groups.

> There is no significant difference in completion time between Scratch users comprehending smelly and non-smelly Scratch programs.

B. Correctness

To calculate the correctness of the answers given by subjects we created an answer model. For tasks T11, T12 and T21 subjects could score 1 point, and for T22 and T23, and T31 to T33 they could obtain 2. For T22 and T23 one point is assigned for correctly answering the first part 'when have you lost or won the game' and one point is assigned for a correct answer to 'what happens then'. For tasks T31 to T33 one point can be obtained for the explanation part and one for the actual implementation. As such, subjects can score a total of 13 points. The highest score achieved by subjects was 12, which two of the subjects achieved, both in the non-smelly group,while the lowest score was zero, scored by only one

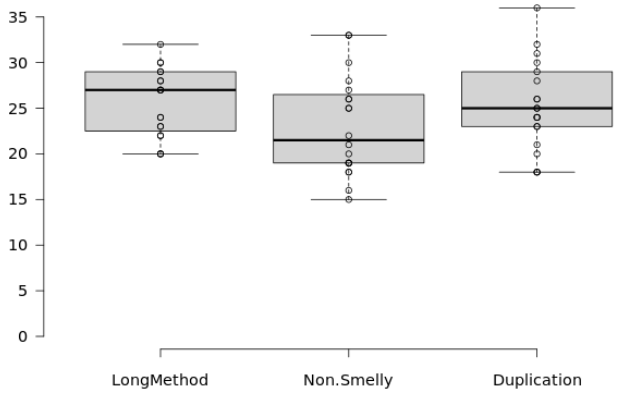

Fig. 8. Time in minutes spent on comprehension tasks (T1 to T33) by three different test groups. Center lines indicate the medians; boxes indicate the 25th and 75th percentiles; whiskers extend 1.5 times the interquartile range from the 25th and 75th percentiles. n = 20, 20, 21 sample points.

subject in the *Long Method* group. Figure 9 represents the correctness data of our experiment for the three test groups. In this figure, as in all other box plots in this paper, white boxes represents groups with a significantly different mean.

We used the Shapiro-Wilk test and found all three sample groups to follow the normal distribution, with means 5.3, 8.2 and 4.9 and standard deviations 2.8, 3.2 and 1.8 respectively. Furthermore, the three groups have equal variance as demonstrated by Levene's test. We thus used the ANOVA test to compare the three samples, which showed that the means of the groups differ significantly ($p<0.05$).

Once we knew about the variation in the three groups, we used pair-wise Student t-tests to compare the groups with each other. Table III shows an overview of the pairwise comparisons. There is a significant difference between performance of students in the non-smelly version of the Pong game, compared to both the *Long Method* and the *Duplication* groups, with effect sizes 0.892 and 1.276 respectively.

There is a significant difference ($p<0.05$) in subject's performance between the non-smelly version of the programs and both the smelly versions.

Table III furthermore shows that there is no difference between the two smell types. Apparently it does not matter for the subjects' performance which of the two smelly versions they received.

There is no significant difference in subject's performance between the *Long Method* and the *Duplication* versions of the program.

TABLE III
P-VALUES AND EFFECT SIZES (COHEN D'S) OF THE PAIRWISE STUDENT-T TESTS BETWEEN THE THREE SAMPLES

	Long Method	*Non-Smelly*
	p-value	
Non-Smelly	0.004	
Duplication	0.238	0.001
	Effect Size (Cohen's d)	
Non-Smelly	0.892	
Duplication	—	1.276

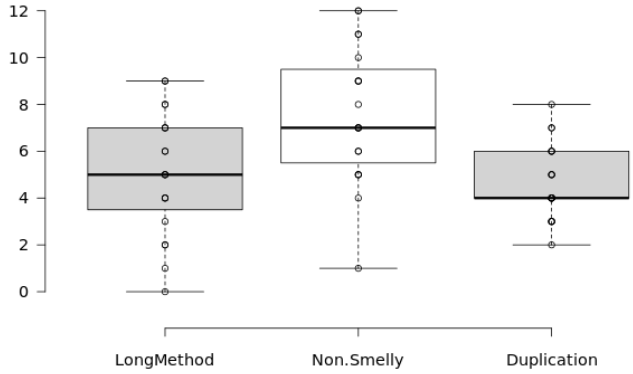

Fig. 9. Number of points obtained by the three different test groups. Center lines indicate the medians; boxes indicate the 25th and 75th percentiles; whiskers extend 1.5 times the interquartile range from the 25th and 75th percentiles. n = 20, 20, 21 sample points.

C. Differences Between the Tasks

To address our third research question, whether certain types of comprehension tasks are disproportionately affected by the two code smells in Scratch, we examine the performance per task in more detail by examining the three categories of questions separately. Figures 10, 11 and 12 present the total scores for each of the three subject groups, for the tasks T11 to T12, T21 to T23 and T31 to T33 respectively.

1) Code Explanation: As hinted on by Figure 10, the differences in performance in the first two tasks are fairly small. We did not measure a significant difference between the three groups for the first three tasks.

We suspect that some subjects were supported by the pictures and the background of the sprites (see Figure 7), reducing the impact of the smelly code. One of the subjects wrote "You make something like ping pong" and another subject said "this looks like Atari BreakOut". Of course, we could have supplied the subjects with just the source code without the pictures, but we wanted to simulate a realistic Scratch code comprehension setting, and within Scratch children have access to all sprites too.

We believe the impact of the pictures was limited however, as many subjects specifically referred to program elements in

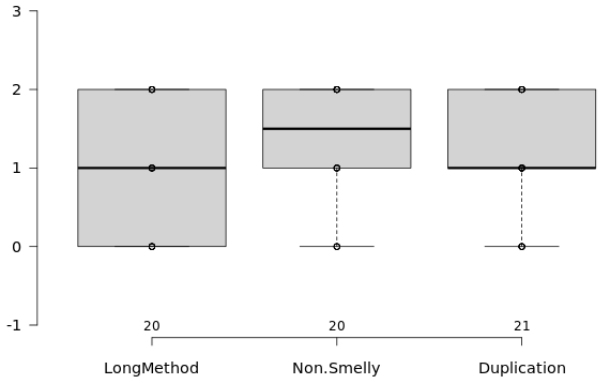

Fig. 10. Number of points obtained by the three different test groups on tasks T11 and T12. Center lines indicate the medians; boxes indicate the 25th and 75th percentiles; whiskers extend 1.5 times the interquartile range from the 25th and 75th percentiles. n = 20, 20, 21 sample points.

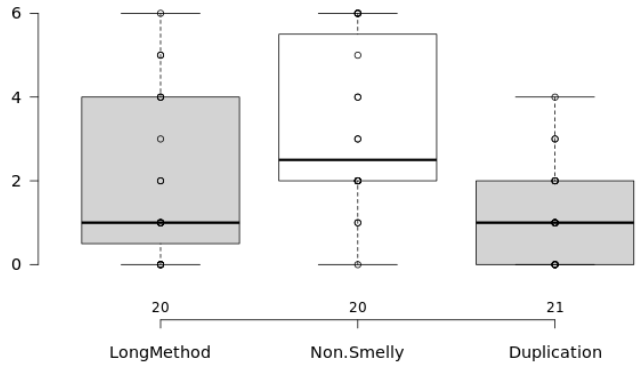

Fig. 12. Number of points obtained by the three different test groups on tasks T31 to T33. Center lines indicate the medians; boxes indicate the 25th and 75th percentiles; whiskers extend 1.5 times the interquartile range from the 25th and 75th percentiles. n = 20, 20, 21 sample points.

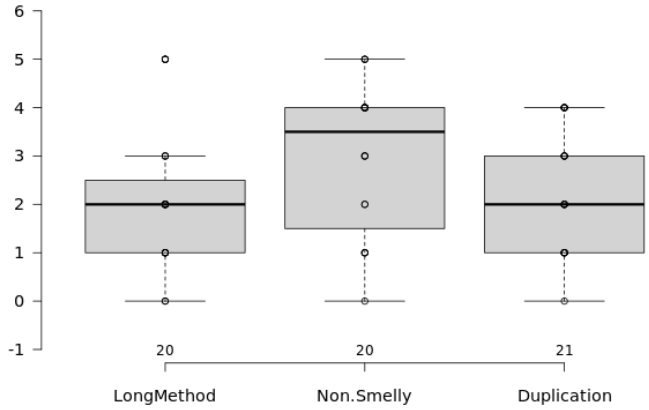

Fig. 11. Number of points obtained by the three different test groups on tasks T21 to T23. Center lines indicate the medians; boxes indicate the 25th and 75th percentiles; whiskers extend 1.5 times the interquartile range from the 25th and 75th percentiles. n = 20, 20, 21 sample points.

their answers, for example one subject explaining the Paddle program saying: "with the one arrow it goes left and with the other it goes right" and another one even specifying "if you press the arrow left, x changes with -25 and if you press right x changes with 15", which does indicate subjects read the Scratch code.

Code explanation is unaffected by the presence of code smells.

2) System Understanding: Figure 11 indicates that in tasks T21 to T23, the tasks pertaining to understanding the entire system, the differences are bigger between the three groups, but these differences are not significant. While there is no significant difference, it seems that *Long Method* performs worst. In the *Long Method* group, 6 subjects out of 20 indicated they did not know what happened in case the game was lost or won (as opposed to answering wrong), while this was only 2 out of 21 for the *Duplication* group and 3 in the Non-Smelly group. Although the differences are small, this seems to indicate that the subjects in the *Long Method* group perceived the program as more difficult. This was corroborated by answers to the reflection question in which we asked what subjects thought of the quality of the programs they worked with. One of the subjects in the *Long Method* group answered the program was "too hard for just clicking on a some arrows". We hypothesize that subjects in the *Duplication* group performed better here because they could find answers to the system understanding tasks in multiple places. For example: the task to answer "when have you won the game" (which is when reaching 5 points) for them could be answered by inspecting any of the three sprites, as they all have events waiting for 5 points are reached.

While not significant, system understanding seems affected more by the *Long Method* smell.

3) System Modification: In the third final category of tasks the subjects had to modify the system, by changing the end of the game to 10 points, and adding text boxes when the game is lost or the Ball touches the Paddle. Here the differences between the groups are large, we measure a significant difference between the three categories (p = 0.002). Comparing the three categories with pairwise Student t-tests, revealed a significant difference between both the non-smelly and the *Long Method* group (p=0.03) and between the non-smelly and the *Duplication* group (p=0.0001). There was a difference in the effect sizes: 0.6 for the comparison of non-smelly with *Long Method* and 1.3 compared to *Duplication* . Thus, the *Duplication* smell affects system modification more heavily.

When looking into the individual results, we observed that 5 out of the 21 participants in the *Duplication* group made typical 'duplication mistakes': they made a change in one of the clones, but not in the other ones. For example,

they changed the "if points = 5" block in the Ball sprite but not in the Paddle sprite, resulting in inconsistent behavior where the winning message is shown at 10 points, but the cheering sound still sounds at 5 points. We observed subjects struggling with this inconsistency in the experiment.

System modification is hampered most by the *Duplication* smell.

VII. Related Work

A. Code Smells

Efforts related to our research include works on code smells, initiated by the work by Fowler [3]. His book gives an overview of code smells and corresponding refactorings. Fowler's work was followed by efforts focused on the automatic identification of code smells by means of metrics. Marinescu [18] for instance, uses metrics to identify *suspect* classes: classes which could have design flaws. Lanza and Marinescu [19] explain this methodology in more detail. Alves *et al.* [20] focus on a strategy to obtain thresholds for metrics from a benchmark. Olbrich *et al.* furthermore investigates the changes in smells over time, and discusses their impact [21]. Moha *et al.* [22] designed the 'DECOR' method which automatically generates a smell detection algorithms from specifications. The CCFinder tool [23] finally, aims at detecting clones in source code, which are similar to our *Duplication* smell.

B. Smells beyond the OO paradigm

In recent times, code smells have been applied to programs outside of the regular programming domain. In our past work, we have, for example, studied code smells within spreadsheets, both at the formula level [5] and between worksheets [4].

More recently, we compared two datasets: one containing spreadsheets which users found unmaintainable, and a version of the same spreadsheets rebuilt by professional spreadsheet developers. The results show that the improved versions suffered from smells to a lesser degree, increasing our confidence that presence of smells indeed coincides with users finding spreadsheets hard to maintain [24].

In addition to spreadsheets, code smells have also been studied in the context of Yahoo! Pipes a web mashup tool. An experiment demonstrated that users preferred the non-smelly versions of Yahoo Pipes programs [6].

C. Quality of Scratch programs

Finally, there are other works on the quality of Scratch programs. There is, for example the Hairball Scratch extension [25], which is a lint-like static analysis tool for Scratch that can detect, for example, unmatched broadcast and receive blocks, infinite loops and duplication. An evaluation of 100 Scratch programs showed that Scratch programs indeed suffer from duplication and bad naming [26].

Most related to our study is the work by Moreno *et al.* [27] who gave automated feedback on Scratch programs to 100 children aged 10 to 14. Their results demonstrated that feedback on code quality helped improve students' programming skills.

VIII. Discussion

A. Threats to validity

There are a number of threats to the internal validity of this study. First of all, some subjects had prior knowledge of Scratch or programming in general. We have reduced this threat by distributing the experienced students over the three groups. Secondly, subjects might have been aware of the goals of the study. We minimized this threat by not revealing to students what type of experiment we were performing, we simply told them after the introductory hour, we wanted to test what they had learned.

There are threats to the external validity of our study too. The generalizability of our results could be impacted by both limited representativeness of the programs and the participating subjects. To mitigate the risks we used programs stemming from the Creative Computing guide designed by the ScratchEd research team at Harvard [14] and we performed the experiment on subjects from three different school classes.

B. Impact on Scratch remixing

Our findings indicate that Scratch users find it harder to interpret smelly programs, especially comprehending Scratch code in the presence of *Long Method* and modifying code with *Duplication* seems to be hard. As Scratch users can inspect and remix other programs as a tool for learning, this remixing can be inhibited by smells in programs. A smell detection or refactoring tool offered to users to improve their code before sharing might increase the ease with which other Scratch users could read and adapt their programs.

C. Smell Detection as a Tool for Education

Originally, smells were designed to improve the code, not the developer. An interesting opportunity of applying code smells to the educational paradigm is to view smells as *learning opportunities*. When a Scratch user inadvertently uses four subsequent move blocks, instead of a loop, this could be an excellent opportunity to introduce the concept of a loop to the user and help them get there from the current code, i.e. perform a refactoring. However the goal in that case is not to improve the maintainability of the code, but to make the user aware of a better way to program, and to introduce them to or remind them of a programming concept.

IX. Concluding Remarks

The aim of this paper is to examine code smells in the context of block-based Scratch programs. As such we have firstly defined two canonical code smells to be applicable to Scratch. We have then evaluated those Scratch smells in a controlled experiment with 61 high-school kids. The results of this evaluation show that subjects performed significantly worse in the presence of the *Long Method* and the *Duplication* code smell, while not affecting the time subjects need

for the assignments. When investigating the different types of tasks in more detail, we observe that *Long Method* mainly decreases subject's understanding of the game as a whole, while *Duplication* makes it harder for subjects to modify their Scratch programs.

The contributions of this paper are as follows:

- The definition of code smells in the context of Scratch programs (Section IV)
- An empirical evaluation of the effect of these smells (Section V,VI)

The current work gives rise to several avenues for future work. Firstly, of course, bigger and longer experiments with a more diverse range of subjects are needed. Would we measure a bigger effect on eight-year olds? Would the effect of code smells remain after children become experienced Scratch programmers? And what about adults like teachers and parents, using Scratch?

Furthermore, we could research a more diverse range of code smells. For example. Scratch code could suffer from the *Temporary Field* smell, when users define variables or signals—user created events—that are not used in the program. Furthermore, Scratch users can define their own programming blocks with parameters, which could lead to the *Many Parameter* smell.

REFERENCES

[1] M. Conway, R. Pausch, R. Gossweiler, and T. Burnette, "Alice: A Rapid Prototyping System for Building Virtual Environments," in *Conference Companion on Human Factors in Computing Systems*, ser. CHI '94. New York, NY, USA: ACM, 1994, pp. 295–296. [Online]. Available: http://doi.acm.org/10.1145/259963.260503

[2] D. Wolber, H. Abelson, E. Spertus, and L. Looney, *App Inventor: Create Your Own Android Apps*, 1st ed. Sebastopol, Calif: O'Reilly Media, May 2011.

[3] M. Fowler, *Refactoring: improving the design of existing code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[4] F. Hermans, M. Pinzger, and A. v. Deursen, "Detecting and Visualizing Inter-Worksheet Smells," in *Proceeding of the 34rd international conference on Software engineering (ICSE 2012)*. ACM Press, 2012, pp. 451–460, to appear.

[5] 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, 2014. [Online]. Available: http://link.springer.com/article/10.1007/s10664-013-9296-2

[6] K. T. Stolee and S. Elbaum, "Refactoring Pipe-like Mashups for End-user Programmers," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 81–90. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985805

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

[8] E. Glinert, "Towards "Second Generation" Interactive, Graphical Programming Environments,," in *Proceedings of the IEEE Workshop on Visual Languages*, 1986.

[9] M. Resnick, J. Maloney, A. Monroy-Hernndez, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai, "Scratch: Programming for All," *Commun. ACM*, vol. 52, no. 11, pp. 60–67, Nov. 2009. [Online]. Available: http://doi.acm.org/10.1145/1592761.1592779

[10] O. Meerbaum-Salant, M. Armoni, and M. M. Ben-Ari, "Learning Computer Science Concepts with Scratch," in *Proceedings of the Sixth International Workshop on Computing Education Research*, ser. ICER '10. New York, NY, USA: ACM, 2010, pp. 69–76. [Online]. Available: http://doi.acm.org/10.1145/1839594.1839607

[11] B. Moskal, S. Cooper, and D. Lurie, "Evaluating the Effectiveness of a New Instructional Approach," in *Proceedings of the SIGCSE technical symposium on Computer science education*, 2005.

[12] S. Cooper, W. Dann, and R. Pausch, "Teaching Objects-first in Introductory Computer Science," in *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '03. New York, NY, USA: ACM, 2003, pp. 191–195. [Online]. Available: http://doi.acm.org/10.1145/611892.611966

[13] T. W. Price and T. Barnes, "Comparing Textual and Block Interfaces in a Novice Programming Environment," in *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ser. ICER '15. New York, NY, USA: ACM, 2015, pp. 91–99. [Online]. Available: http://doi.acm.org/10.1145/2787622.2787712

[14] K. Brennan, C. Balch, and M. Chung, *CREATIVE COMPUTING*. Harvard Graduate School of Education, 2014.

[15] C. Lange and M. Chaudron, "Interactive Views to Improve the Comprehension of UML Models - An Experimental Validation," in *15th IEEE International Conference on Program Comprehension, 2007. ICPC '07*, Jun. 2007, pp. 221–230.

[16] J. Quante, "Do Dynamic Object Process Graphs Support Program Understanding? - A Controlled Experiment." in *The 16th IEEE International Conference on Program Comprehension, 2008. ICPC 2008*, Jun. 2008, pp. 73–82.

[17] M. Pacione, M. Roper, and M. Wood, "A novel software visualisation model to support software comprehension," in *11th Working Conference on Reverse Engineering, 2004. Proceedings*, Nov. 2004, pp. 70–79.

[18] R. Marinescu, "Detecting Design Flaws via Metrics in Object-Oriented Systems," in *Proceedings of TOOLS*. IEEE Computer Society, 2001, pp. 173–182.

[19] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*, 2006. [Online]. Available: http://www.springer.com/alert/urltracking.do?id=5907042

[20] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *26th IEEE International Conference on Software Maintenance (ICSM 2010)*. IEEE Computer Society, 2010, pp. 1–10.

[21] 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 International Symposium on Empirical Software Engineering and Measurement*. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 390–400.

[22] N. Moha, Y. Guhneuc, L. Duchien, and A. Le Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, Jan. 2010.

[23] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *TSE*, vol. 28, no. 7, Jul. 2002.

[24] B. Jansen and F. Hermans, "CODE SMELLS IN SPREADSHEET FORMULAS REVISITED ON AN INDUSTRIAL DATASET," in *Proceedings of the International Conference on Software Maintenance and Evolution*, Bremen, Germany, 2015, pp. 372–380.

[25] B. Boe, C. Hill, M. Len, G. Dreschler, P. Conrad, and D. Franklin, "Hairball: Lint-inspired Static Analysis of Scratch Projects," in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '13. New York, NY, USA: ACM, 2013, pp. 215–220. [Online]. Available: http://doi.acm.org/10.1145/2445196.2445265

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

[27] J. Moreno-Len, G. Robles, and M. Romn-Gonzlez, "Dr. Scratch: Automatic Analysis of Scratch Projects to Assess and Foster Computational Thinking," *RED : Revista de Educacin a Distancia*, no. 46, pp. 1–23, Jan. 2015. [Online]. Available: https://doaj.org