Habits of Programming in Scratch

Orni Meerbaum-Salant Michal Armoni Mordechai Ben-Ari
Department of Science Teaching
Weizmann Institute of Science
Rehovot 76100 Israel
{orni.meerbaum-salant,michal.armoni,moti.ben-ari}@weizmann.ac.il

ABSTRACT

Visual programming environments are widely used to introduce young people to computer science and programming; in particular, they encourage learning by exploration. During our research on teaching and learning computer science concepts with Scratch, we discovered that Scratch engenders certain *habits of programming*: (a) a totally bottom-up development process that starts with the individual Scratch blocks, and (b) a tendency to extremely fine-grained programming. Both these behaviors are at odds with accepted practice in computer science that encourages one: (a) to start by designing an algorithm to solve a problem, and (b) to use programming constructs to cleanly structure programs. Our results raise the question of whether exploratory learning with a visual programming environment might actually be detrimental to more advanced study.

Categories and Subject Descriptors

K.3.2 [**Computers & Education**]: Computer and Information Science Education - *Computer Science Education.*

General Terms

Human Factors.

Keywords

Scratch, middle schools, habits

1. INTRODUCTION

Scratch [11] is a visual programming environment that is widely used by young people. Scratch is used by individuals for self-study outside of any educational framework; it is used in informal settings like clubs and summer camps [8]; and, it is used in schools at all levels. Even lecturers at universities have taken to using Scratch in CS1 courses before plunging into programming in professional languages [7].

While the attractiveness of the Scratch environment and its ability to motivate young people are widely attested, we are interested in exploring whether Scratch can be used to teach *concepts* of computer science and programming. In previous work [6, 9], we showed that learning by middle-school students is uneven at best. Contrary to the claim that open-ended exploration can achieve satisfactory learning outcomes [2], we found that concepts were only learned when students were explicitly taught the concepts while they created projects that use the concepts. This is not intended to denigrate Scratch in any way, but rather to emphasize that Scratch is just a tool, and that good teaching methods and learning materials are required to maximize potential learning with the tool.

During our research on learning concepts, we found incidentally that Scratch influenced not only the learning of concepts but also the *habits of programming* that the students develop. In this paper, we present the habits of programming that we found and we attempt to explain their development within the Scratch environment. Since these habits are very much at odds with the accepted practice, our research raises the possibility that learning with Scratch could be detrimental to successfully learning programming.

Section 2 contains a brief overview of Scratch and previous work on habits. Our research methodology is described in Section 3. Section 4 presents the two primary classes of programming habits that we found. Section 5 discusses the results and attempts to provide explanations for them. In the concluding Section 6, we reflect on the implications of our results and offer suggestions for further research.

2. BACKGROUND

2.1 The Scratch Environment

Scratch is a visual programming environment that was developed by the Lifelong Kindergarten group at the MIT Media Laboratory. Scratch is intended to foster creativity, increase motivation to engage with computers and reduce the anxiety that can result from the engagement. Like its predecessor LOGO, Scratch is based on constructionism [2, 5]. Programming is done by dragging and dropping blocks to form scripts that control the animation of two-dimensional sprites on a stage. The elimination of syntax errors makes Scratch accessible to young people, and most users create colorful games and stories. As reported in the literature, Scratch encourages self-directed learning: Many users learn Scratch as they go, trying commands from the blocks palette and using code from existing projects [8].

2.2 Habits and programming

Joni and Soloway [4] argued that educators cannot be satisfied when students produce programs that "just work." They found that students may write correct programs in the sense that they work (that is, they have correct I/O behavior for all input from the problem space), but cannot be considered as good since they are poorly structured. They recommended teaching students that readability (and therefore, good structure) is a criterion according to which programs are evaluated. Similarly, in mathematics, Cuoco et al. recommended that students be taught good habits of problem solving [1], and that these should even serve as a theme around which to organize the curriculum.

There are several websites that list good habits that programmers should have, in particular, ones that concern configuration management, testing and documentation:

http://web.mit.edu/~axch/www/programming_habits.html

http://drupal.technicat.com/writing/programming.html.

HabiPro (Habits of Programming) is "a pedagogical and collaborative software designed to develop good programming habits. It doesn't try to teach programming but to develop in the novice student skills such as observation, reflection or structure, which are necessary to become good programmers" [13].

All this points out that an examination of teaching and learning processes in computer science should not neglect the aspect of programming habits.

3. RESEARCH METHODOLOGY

The phenomenon described in this paper arose during an investigation into the learning of computer science concepts in Scratch [8]. That research methodology was designed with certain goals in mind, but, serendipitously, during the data collection and its qualitative analysis, interesting findings arose, which we report on in this paper. In this section, we summarize those aspects of the research methodology of [8] that are relevant to this article.

We studied novices who used the Scratch environment in middle schools. Our subjects were drawn from two classes: one consisted of 18 students (11 boys and 7 girls), while the other consisted of 28 students (all boys, studying in a boys-only school). The students in both classes were 14–15 years old (ninth grade). Each class took place in one two-hour period a week for one semester.

The teacher of the first class taught mathematics in middle school and had no CS teaching experience, while the teacher of the second class had 15 years experience teaching CS. Both were encountering the Scratch environment for the first time.

A draft of a textbook written by the second and third authors was available to the teachers, but not to the students. This book emphasizes the process of developing a program by posing an algorithm, designing a solution and only then implementing the solution in the programming environment.

The investigation described here is based primarily on three sources of qualitative data:

- The first author was a non-participant observer in both classes, observing the students as they solved problems in class for two hours a week during the entire school year. The observations were documented in field notes.
- The students' work was collected and analyzed, including solutions to exams. In addition, 34 projects that they submitted for presentation at a public Scratch Day held at our institution were collected and analyzed.
- Interviews were conducted with ten students and two teachers. In addition, a discussion was held with a focus group consisting of two students from one class.

We emphasize that these tools were not designed to document the phenomena described in this paper; rather, these phenomena arose from the data, appearing repeatedly in different types of data.

4. HABITS OF PROGRAMMING

According to a dictionary (http://merriam-webster.com), a *habit* is "a settled tendency or usual manner of behavior" or "an acquired mode of behavior that has become nearly or completely involuntary." Our observations of students' behavior when programming with Scratch, together with our analyses of the programs they developed, led us to identify *habits of programming*, that is, habits used in the process of solving programming tasks.

The two characteristics we looked for in order to identify a habit were: (1) a behavior must be "settled" or "usual," in the sense that the behavior appeared in the work of many students, as well as on numerous occasions for an individual student; (2) the behavior must be "involuntary," in the sense that the students demonstrated the behavior unconsciously without attempting to justify it and without considering alternatives.

We identified two programming habits that were demonstrated over and over again during the students' work and in the resulting projects. The following subsections describe these two programming habits. To simplify the presentation and to stay within the limits of a conference paper, we will use a few concrete examples to exemplify the habits, although we found many more in our analysis of the data.

4.1 Bottom-up programming

In a bottom-up programming approach one starts with components, which are then linked together to form a larger subsystem, until a complete top-level system is formed. When used correctly, a bottom-up approach enables a programmer to design, implement and test logically-coherent components that can then be integrated to form a software system. In our case, students took this approach to its extreme, starting with the most basic elements of Scratch, the blocks with the instructions. During the observations and the interviews, we saw that when faced with a programming task, the students did not approach it by thinking on the algorithmic level and not even on the level of software design. Instead, they began to solve a problem by dragging all the blocks that seemed to be appropriate for solving the task, and then combining them into a script. This pattern of behavior can be characterized as programming by *bricolage*, as advocated by Turkle and Papert [12].

The Scratch environment fosters the development of this habit: All the instructions are given on the blocks palette that is visible at all times, so the users need not remember the instructions nor need they deliberate as to what instructions are needed. This is further exacerbated by the fact that individual instructions or fragments of scripts can be left in the script area without affecting the computation of the program that is executed. While this aids the interactive construction of scripts, it is obviously conducive to bricolage.

This habit was described by a student during an interview:

> When I need to solve a programming problem, first of all I choose which instructions to drag and drop to the script area, and then I try to see how all the instructions will fit together in the best way, what will be the simplest way to solve the problem without any interruption or difficulties.

4.2 Extremely Fine-Grained Programming

The second programming habit that we found complements the first one, in that it takes the top-down approach to its extreme. In a top-down approach [3], tasks are decomposed into smaller, more tractable subtasks. When used correctly for designing software, the decomposition is into logically coherent units that facilitate development and improve maintainability.

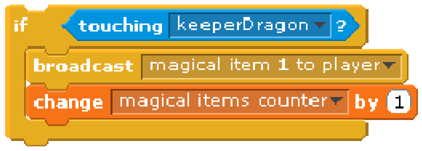
When analyzing the students' artifacts, we saw that they carried out the decomposition until the units (the scripts) became extremely small, and usually lacked logical coherency. We call this *extremely fine-grained programming (EFGP).*

4.2.1 An example of EFGP

We will use the following example to demonstrate this habit: In a game, the player collects magical items by fighting their guards. Every time the player hits a guard (by touching him with his sword), he obtains the magical item that was watched over by the guard. When the player has collected six items, he can move to the next level.

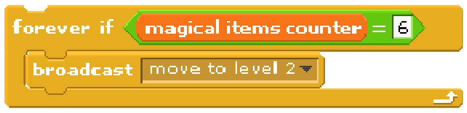
A script to handle the event of the player winning a fight should be composed of the following steps: (a) move the item to the player's bag (by sending an appropriate message to the item); (b) update a counter of the items in the bag; (c) if the counter reaches six, move to the next level.

Here is an EFGP implementation of this sequence of steps:[1]



```
if touching keeperDragon ?
    broadcast magical item 1 to player
    change magical items counter by 1
```

In Script1



```
forever if magical items counter = 6
    broadcast move to level 2
```

In Script2

Although the three steps *as a whole* form a logically coherent unit, this student decomposed it further, creating a separate script for the third step of deciding whether to move to the next level. Furthermore, this script was for a *different* sprite, one that had nothing at all to do with the event of winning a fight!

As a result of the habit of EFGP, students' projects contained a very large number of scripts (occasionally hundreds).

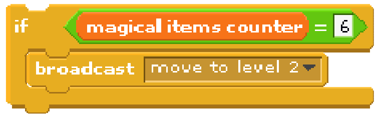
[1] The examples we use are taken from students' projects, but have been simplified to obtain a concise and clear presentation.

Scratch is to be praised for its clear and convenient support of decomposition into multiple scripts for multiple sprites. However, it seems likely that this ease of decomposition fosters the habit of EFGP. The habit is closely related to habit of extreme bottom-up programming described in Section 4.2, in the sense that both demonstrate the lack of a design phase during the development.

In the following subsections, we analyze the habit of EFGP in more detail, relating it to specific concepts computer science.

4.2.2 EFGP and control structures

When the decomposition is very fine-grained, the use of control structures is affected in the sense that they are not always used as they should be and sometimes are not used at all. For example, the simplest implementation of the third step of the algorithm presented above uses a conditional statement::

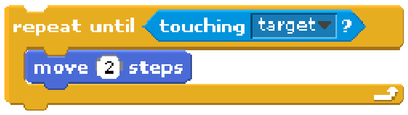


```
if magical items counter = 6
    broadcast move to level 2
```

However, the student's solution used a conditional infinite loop, turning a simple conditional into a busy-wait loop. This phenomenon—the reduced use of if-blocks—was frequently found in the students' projects.

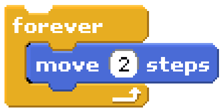
The reduced used of conditional execution in EFGP carried over to the more complex `if <cond> do <op1> else do <op2>` construct, which aggregates two subtasks together. Students tended to decompose this construct into the two smaller constructs, one for each subtask: `if <cond> do <op1>` and `if <not cond> do <op2>`.

The use of loop structures was similarly affected. A (finite) loop is a control structure that encapsulates two subtasks: repetition of a sequence of instructions and termination of the repetition when appropriate (after a certain number of times or when a certain condition holds). EFGP decomposes these two subtasks, resulting with an infinite loop that implements the repetition task, while another script handles the ending of the loop! Consider, for example, a game in which a missile moves until it touches a target. A simple control structure implementing this subtask might be:



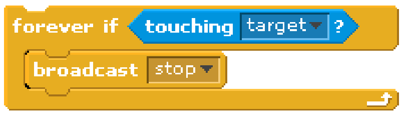
```
repeat until touching target ?
    move 2 steps
```

Here is an EFGP implementation:



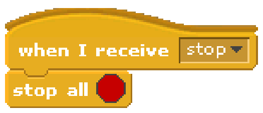
```
forever
    move 2 steps
```

In Script1



```
forever if touching target ?
    broadcast stop
```

In Script2



In Script3

The decomposition of the simple loop is into *three* subtasks: repeatedly executing the move instruction, checking the termination condition and then terminating the execution. We found this frequently: EFGP resulted in a reduced use of finite looping structures so that for-loops and repeat-until loops were replaced by forever-loops with external control of the execution.

4.2.3 EFGP and structured programming

Not only did EFGP result in a skewed use of the various control structures, but it also resulted in programs that we judged qualitatively to be poorly structured. A repeat-until loop is a coherent logical concept where the body of the loop and the condition for its termination are co-located and easy to understand. When these components of the loop construct are no longer co-located, it becomes very difficult to read and understand a program. This can be seen as analogous to objections to the goto-instruction, which can cause unstructured (“spaghetti”) programs that have lost their logical coherence. Defenders of the goto-instruction claimed that the instruction was not at fault and that the instruction could be used in ways that were not detrimental to the structure of a program, but the consensus in modern language design and programming is that structured constructs like repeat-until loops should always be used *unless there is a special reason not to*. The students did not justify their choice of structures, though they are certainly too inexperienced to do so. Similarly, the forever-instruction should not be blamed for “spaghetti” code in Scratch; instead, one should look for reasons why the control structures were not used in the ways they were designed to be used.

4.2.4 EFGP and concurrency

The Scratch environment encourages the use of concurrency since all scripts of all sprites are executed at the same time. Extreme decomposition necessarily results in a highly concurrent program, but one in which the concurrency was *not consciously designed*. When actions are executed concurrently, understanding the execution is not simple since actions may be interleaved in various ways, leading sometimes to unexpected, even unwanted, results. Indeed, our students were frequently helpless when faced with unanticipated problems caused by concurrency issues.

For example, in the project described in Sction 4.2.1, the move to the next level might not happen when the student wanted it to happen. After the value of the counter became 6 in Script1, the player might touch the guard again, increasing the value of the counter to 7 before the condition in Script2 was checked. This would not have happened had the event of winning been handled in one script, containing the three-step sequence described above.

Since concurrent scripts (both within a single sprite and in separate sprites) are such an integral part of Scratch, one cannot avoid this issue when teaching Scratch. The textbook presented concurrency as early as in the second chapter, but in an informal manner. Synchronization of concurrent scripts is a very difficult concept, so a more formal and complete treatment was deferred to a chapter near the end of the textbook. Neither class had time to learn this material, and, in any case, it is unreasonable to assume that young novice students will easily develop the skills necessary to debug concurrent programs (even assuming that "debugging" is a viable concept in the context of concurrency).

Unfortunately, EFGP exacerbates the problem since the plethora of scripts makes problems more likely to occur. Given the very large number of scripts in the students’ projects, race conditions were very common. The massive concurrency that results from EFGP made the programs difficult to debug and we believe that had the students developed programs using fewer, logically coherent, scripts, the programs would have been much easier to understand and debug.

5. DISCUSSION

We identified two habits of programming demonstrated by students who worked with the Scratch environment. Both of these habits are at odds with the accepted practice of computer science. Since habits tend to be persistent, this raises the possibility that they will be retained as students advance from an educational visual programming environment like Scratch to professional languages and environments.

The bottom-up programming habit is clearly encouraged by the characteristics of the Scratch environment and is in line with Papert's philosophy of constructionism [2] and with bricolage [12]. Normally, one would not be surprised that program design did not take place if no design is taught, but in our case, we did try to do so and still the results were not satisfactory from our point of view. As noted above, our textbook *does* emphasize analysis and design. Furthermore, one of the teachers was an experienced high-school teacher of computer science, who pays careful attention to teaching design in her courses at that level. Why, then, did she not emphasize program design in the context of this course, when clearly she was capable of doing so and had the support of the textbook?

When asked, the teacher agreed that she is fully familiar with the importance of program design. She claimed that her inexperience with Scratch was the reason that she did not engage the students in design during the teaching process. We believe that another factor may be relevant here: the colorful interface of Scratch and the fun of creating animated games can give the impression that Scratch is a toy or a video game; this has the potential to cause teachers to relax their vigilance concerning software design during teaching process. However, for all its glamour, Scratch is a sophisticated programming environment, and we believe that it should be treated like any other programming environment: as a tool with which to teach sound habits of programming. These sound habits do not develop by themselves; they can only develop if diligently instilled by the teacher.

The habit of EFGP is characterized by decomposition into very small, incoherent, modules. Modularity is a fundamental principle of software design, and indeed the textbook emphasizes decomposition into subtasks. However, the extreme to which this principle is taken results raises a few concerns.

First, we are concerned by the incorrect use of control structures. One major objective of an introductory course is to expose novice students to fundamental ideas such as algorithmic control structures. We are especially disturbed by the fact the students avoided the use of the most important structures: conditional execution and bounded loops. It is uncontroversial that these are difficult to learn, so it is unfortunate that students miss the opportunity to learn these structures in a fun environment.

We may be partly to blame because our textbook teaches the simpler if-statement before the if-then-else-statement and the simpler forever-loop before the repeat-until loop and the for-loop. This seems sound from a pedagogical point of view, but it does demand that the teaching process emphasize the more complex constructs and encourage their use in preference to the EFGP style of programming. A similar consideration applies not just to the individual control structures, but also to the inability of the students to write a well-structured program, a skill that is central to CS education.

An important advantage of the Scratch environment is that it lends itself naturally to projects such as games, which the students are able to implement themselves. But creating games by extremely fine-grained programming leads to projects with hundreds of concurrent scripts that are practically impossible for the students to debug and maintain. Paradoxically, the motivation that results from the ability to program interesting games can dissolve when the debugging process becomes difficult and frustrating!

6. CONCLUSION

While we are pleased with the willingness of students to engage in programming by using Scratch and with the technical skills that they develop, we are disturbed by the habits of programming that we uncovered. These habits are not at all what one expects as the outcome of learning computer science. Any habit, including a programming habit, tends to be persistent, so it is possible, even likely, that these bad habits will transfer to the students' further CS studies. On the other hand, perhaps they can outgrow these bad habits.

Our results can be framed as a dilemma: should we make things "easy" for students during their initial studies or should we teach them the "right way" from the beginning? This dilemma is extremely common in computer science education, because it arises any time an educational language, environment or technique is proposed. For example, even the "objects-first" controversy can be framed as a dilemma between learning what some consider as "the right way" initially vs. learning it later when the students have more experience and are ready to understand it.

This is not a question that can be answered by debate; instead, it is an empirical question that needs to be elucidated with further research (both qualitative and quantitative).

7. ACKNOWLEDGMENTS

This research was partially supported by the Israel Science Foundation grant 09/1277 and by a Sir Charles Clore Postdoctoral Fellowship.

8. REFERENCES

[1] Cuoco, A., Goldenberg, E.P., and Mark, J. 1997. Habits of mind: An organizing principle for mathematics curriculum. *Journal of Mathematical Behavior*, 15(4), 375-402.

[2] Harel, I., and Papert, S. (eds.). 1991. *Constructionism.* Ablex, Norwood, NJ.

[3] Hartman, J., 1991. Understanding natural programs using proper decomposition, *Proceedings of the 13th International Conference on Software Engineering* (Austin, Texas, May 13-17, 1991), 62-73.

[4] Joni, S. A., and Soloway, E., 1986. But my program runs! Discourse rules for novice programmers. *Journal of Educational Computing Research*, 2(1), 95-126.

[5] Kafai, Y., and Resnick, M., (eds.) 1996. *Constructionism in Practice: Designing, Thinking, and Learning in a Digital World*. Lawrence Erlbaum Associates, Mahwah, NJ.

[6] Kaloti-Hallak, F., 2010. *Learning Programming Concepts Using Scratch at the Middle-School Level*. Unpublished MSc Thesis, Weizmann Institute of Science.

[7] Malan, D. J., and Leitner, H. H., 2007. Scratch for budding computer scientists. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '07)*. ACM, New York, 223-227.

[8] Maloney, J.H., Peppler, K., Kafai, Y., Resnick, M., and Rusk, N. 2008. Programming by choice: Urban youth learning programming with Scratch. *SIGCSE Bull.* 40, 1 (March 2008), 367-371.

[9] Meerbaum-Salant, O., Armoni, M., and Ben-Ari, M., 2010. Learning computer science concepts with Scratch. In *Proceedings of the Sixth International Workshop on Computing Education Research (ICER '10)*. ACM, New York, 69-76.

[10] Papert, S., 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, New York.

[11] Resnick, M., Maloney, J., Monroy-Hernandez, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., and Kafai, Y., 2009. Scratch: Programming for all. *Commun. ACM* 52, 11 (November 2009), 60-67.

[12] Turkle, S., and Papert, S., 1991. Epistemological pluralism and the revaluation of the concrete. In: Harel, I. and Papert, S. (eds.), Constructionism. Ablex, Norwood, MA, 161–192.

[13] Vizcaino, A., Contreras, J., Favela, J., and Prieto, M. 2000. An Adaptive, Collaborative Environment to Develop Good Habits in Programming. In *Proceedings of the 5th International Conference on Intelligent Tutoring Systems (ITS '00)*, Gilles Gauthier, Claude Frasson, and Kurt VanLehn (eds.). Springer-Verlag, London, UK, 262-271.