Scratching the Surface of Advanced Topics in Software Engineering: A Workshop Module for Middle School Students

Paolo A. G. Sivilotti and Stacey A. Laugel
Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{paolo,laugel}@cse.ohio-state.edu

ABSTRACT

A common approach for introducing computer science to middle school students is to teach them a simple yet engaging programming language. A different approach is to teach them some advanced topic independent of any particular language or syntax. We describe a 3-hour workshop module designed to do both. This module has been piloted with a group of thirty 8th grade girls. It uses the Scratch programming language to develop the advanced software engineering concepts of specifications, refinement, and composition. After this module, students were enthusiastic about continuing to program in Scratch independently and also felt they learned something about computer science as a discipline.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education*

General Terms

Design, Human Factors

Keywords

K-12 outreach, specifications, refinement, Scratch

1. INTRODUCTION

A variety of programming languages and environments have been designed explicitly as first programming languages, often targeting a K-12 audience. Examples include Logo, Karel the robot, LEGO Mindstorms, Phrogram, Alice, and Scratch. Although these environments differ in their particulars, they are similar in their focus on (i) student engagement and (ii) core computational concepts such as sequencing, iteration, and conditionals. Thus, these languages can be an effective introduction to the science of computing.

SIGCSE'08, March 12–15, 2008, Portland, Oregon, USA.

Furthermore, as students gain experience in programming, they eventually encounter more advanced concepts in algorithms, encapsulation, design, and abstraction. Indeed, many of these languages have been used beyond K-12, for introductory courses at the college level [6, 5].

A structured first exposure to computer science often occurs as part of a school or camp curriculum. In such a setting, there is time for students to learn a simple language, explore and create programs using that language, and gradually advance through more complex constructs and issues.

Not all structured first exposures, however, have the benefit of an extended (or even repeated) interaction with students. For example, every summer The Ohio State University hosts a week-long day camp for middle school girls to introduce them to various science and engineering disciplines. One of the authors has organized the computer science module as part of this camp since its inception in 2002. The primary challenges in organizing such a module are the limitation in time (2 to 3 hours) and the lack of programming experience amongst participants.

For an isolated workshop module such as this one, there are few natural choices. One option is to introduce an elementary programming environment. Students learn some basic computational concepts, and—more importantly—can become engaged enough that they continue programming independently after the workshop is over.

Another option is to expose students to some advanced concepts in computer science. Students gain an appreciation for the intellectual profile of the discipline which may resonate on a more fundamental level than any small-scale programming activity. Examples of such an approach include the Groupthink activity to teach specifications [2], kinesthetic learning activities to teach distributed self-stabilization [4], and "unplugged" activities to teach algorithms [3].

For a single short workshop module, these two options may appear to be mutually exclusive. One either introduces a programming language/environment, or an advanced concept without any programming. However, in summer of 2007, we developed and conducted a workshop module for middle school girls to acheive *both* goals. We were inspired by personal anecdotal evidence that, for this audience, Scratch is a highly engaging programming environment and requires relatively little hands-on instruction. Furthermore, we were confident that this audience could appreciate advanced concepts in computer science given our earlier success with teaching self-stabilization to middle school students [4].

This 3-hour workshop module assumes no previous programming experience. The module includes both lectures and hands-on programming activities. The activities are designed to introduce students to several advanced concepts in software engineering, specifically: pre- and post-condition specifications, refinement of specifications, and composition. The goal is for students to appreciate some of the intellectual foundations of computer science, while at the same time becoming engaged enough with Scratch to want to continue programming on their own.

We evaluated the effectiveness of our workshop module relative to a control group in which only Scratch programming was introduced (i.e., without advanced concepts). Based on retrospective self-evaluations, the advanced topics group (i) felt they learned more than the control group, and (ii) were more likely to download Scratch at home to continue programming independently.

2. BACKGROUND

2.1 Scratch

Scratch is a visual programming environment in which users create programs by dragging and dropping individual action blocks that then snap together into scripts. The language emphasizes visual effects and sound, allowing users to easily create sprites, each of which is controlled by its own set of scripts.

The design of Scratch deliberately favors novice programmers, emphasizing simplicity and ease-of-use over functionality and power. For example, although sprites bear some resemblance to objects, there is no notion of classes or object instantiation. To make multiple instances, a sprite must be copied and pasted. Similarly, the language does not include constructs corresponding to inheritance, generic types, or interfaces.

The exclusion of advanced constructs directly contributes to Scratch's success at its chosen role: an engaging first programming language that enables the expression of one's creativity through computation and multimedia. Because of this simplicity, students can quickly understand enough to write interesting programs. That is, Scratch affords the opportunity for a positive programming experience practically "out of the box". Furthermore, the absence of more exotic (and powerful) features means that even a novice programmer can understand practically any Scratch program, at least in terms of its constituent parts.

Of course, these same qualities that make Scratch so effective as a first programming language can also limit its utility for teaching advanced concepts. Using Scratch in this role (as is done in the module presented in this paper) should be undertaken with considerable caution (and perhaps some healthy apprehension).

2.2 Learning Objectives

The learning objective of this module is exposure to (not proficiency in) the following advanced concepts in software engineering:

Programs vs. Specifications. Both encode a mapping from initial to final states. Programs give specific instructions for *how* to carry out this transformation from input to output, while a specification describes *what* the relationship is between the two.

Refinement: Strengthening the precondition and weakening the postcondition. Hoare's rule of consequence states that specifications are *antimonotonic* in their precondition and *monotonic* in their postcondition. Put another way, the more information is given about inputs (and the less information is given about outputs), the easier it is to implement a specification.

Composition. The output from one program can be consumed as input by another program. In this way, smaller, simpler programs can be hierarchically composed into more complex systems.

In addition to these advanced topics, it is also hoped that students leave with the sense that programming is fun and is something they can do on their own.

3. MODULE OVERVIEW

The module consists of an introductory lecture, followed by two programming labs. The first lab develops the notion of specifications and refinement, while the second deals with composition. Pair programming is used for each lab.

The three hour time slot breaks down as follows:
20 min – Introductory lecture
45 min – Lab 1 activity
20 min – Lab 1 debrief and mini-break
20 min – Lab 2 activity part (a)
15 min – Break
10 min – Lab 2 discussion
45 min – Lab 2 activity part (b)
5 min – Module debrief

A unifying analogy is used throughout the module: programs as recipes. This analogy provides an informal, familiar context for presenting both the basic and advanced concepts.

4. INTRODUCTORY LECTURE

Since no prior programming experience is expected for this module, an introductory lecture is needed to give the students some basis for carrying out the labs and for appreciating the advanced topics underlying the activities.

This short lecture (15 minutes) develops the key analogy used throughout the module: a computer as a chef, and a program as a recipe. This is a common analogy in introductory courses, as exemplified by the classic PB&J sandwich activity [1]. In our case, though, the point of this analogy is not how things can go wrong if there are ambiguities in the program.[1] Instead, the point made is that a chef is a general-purpose processor, capable of transforming ingredients (input) into a final dish (output), while the recipe is the sequence of instructions for *how* to carry out this transformation. Software engineering, then, is cast as "recipe engineering".

After defining programs by analogy, we then discuss requirements and their role in the broader context of engineering. Example requirements are given from civil engineering

[1]In our experience, such a lesson is entirely counterproductive and misplaced as an early introduction to computer science. Rather than focusing on disastrous—though perhaps humorous— outcomes resulting from incorrect programs, it seems more appropriate to engage students with the exciting, creative possibilities enabled by writing *correct* programs.

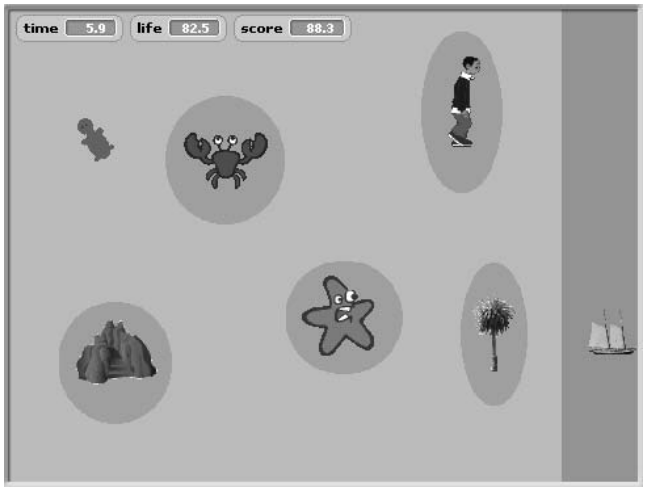

Figure 1: Stage for "Save the Turtle" Lab

where a bridge is required to span a certain distance and carry a certain load. A clear distinction is made between requirements and designs. The former is provided by the client, the latter by the engineer. The former defines what it means for the latter to be correct. Engineering is cast as a problem-solving discipline in which a design must be developed that is both "correct" (meets its requirements) and "good" (based on metrics such as cost, weight, etc).

To bring the discussion back to software engineering, the question is posed: "What kinds of requirements would be given to a recipe engineer?" Students quickly see that the final dish is an important part of the specification. To elicit a more complete view, the point can be made that there is a trivial recipe for solving the problem if only the final dish is listed: Use a single ingredient, the final dish itself. The conclusion, then, is that recipes (programs) are specified by both their ingredients (inputs) and final dish (output).

At the end of this lecture, a very brief introduction is given to Scratch. No more than 5 minutes are needed to demonstrate the user interface, some basic types of blocks, the stacking of blocks, and the creation of sprites.

5. LAB 1: PROGRAMS, SPECIFICATIONS, AND REFINEMENT

In the first lab, students are asked to solve a series of progressively harder programming tasks. They are given a Scratch program consisting of a turtle sprite and several obstacle sprites on a beach. For each task, students write a script to navigate the turtle safely (i.e., without hitting obstacles) across the beach (see Figure 1).

The tasks differ in the conditions under which this navigation must be accomplished:

(A) Fixed start / Reach the ocean. The turtle always begins in the same position in the upper left-hand corner and facing the same direction; it must reach the ocean along the right-hand side of the stage.

(B) Random-facing start / Reach the ocean. The turtle always begins in the same position as in (A), but facing any random direction; it must reach the ocean as in (A).

(C) Fixed start / Reach the ship. The turtle begins as in (A) but must reach the ship floating on the ocean.

(D) All-random start / Reach the ocean. The turtle can begin in any position and facing any direction; it must reach the ocean.

In order to rule out trivial solutions, students are prohibited from modifying the scripts of any sprites other than the turtle. In order to guarantee that collisions with obstacles are detected, students are also prohibited from using "goto x/y" and "set x/y" blocks and from moving more than 10 steps with a single block.

Most teams completed the first three tasks fairly easily, since these can be accomplished with straight-line scripts. Task (D), however, requires a conditional and sensing the proximity of an obstacle. Some teams discovered Scratch's conditional control blocks on their own while the rest were given a hint after struggling for a while.

A small element of competition was added to this lab by including a score that decreased with time and the number of obstacles encountered along the journey. Teams that finished early were encouraged to optimize their score.

In the debriefing following this lab, students had no difficulty in agreeing that (A) was the easiest task and (D) was the most difficult. When asked specifically why (C), reaching the ship, was more difficult than (A), students immediately saw that the extra constraint on the result meant more work for the programmer. The students also recognized that the random facing start in (B) was more difficult than (A) because less information was given about the input. The recipe analogy was used as a further illustration of this principle. Students considered the relative difficulty in writing a recipe for chocolate chip cookies, for any kind of cookie, for anything sweet, or for anything edible. Again, the more constrained the final dish (output), the harder the recipe (program) is to write.

Finally, the corresponding observation was made for preconditions. Saving the turtle by reaching the ocean is easier when both initial position and direction are known, slightly harder when only initial position is known, and harder still when neither is known. In terms of recipes, students considered the relative difficulty in writing a recipe for chocolate chip cookies given 1/2 cup of unsalted butter, given 1/2 cup of some kind of fat, or given some amount of some kind of fat. The more is known about the ingredients (input), the easier the recipe is to write.

6. LAB 2: COMPOSITION

The second lab is subdivided into two parts. In the first part, each team is asked to create a sprite that meets a specific set of requirements. In the second part, teams import the sprites written by other teams to form a complete Scratch project. The theme for this lab is "dragons and butterflies" since the assembly of the sprites written by different teams results in an interactive game in which fire-breathing dragons chase fluttering butterflies and try to scorch them.

In the first part of the lab, teams are given a precise description of the required behavior for one of the following sprites:

- A flying dragon that moves around in response to the arrow keys being pressed.

- A fire-breathing dragon that breathes fire when the space bar is pressed.
- A fluttering butterfly that flies around in a random pattern.
- A scorched butterfly that appears to burn when it touches a particular shade of red.
- Timer and score-keeper sprites that announce the end of the game when too much time has expired (a loss) or enough butterflies have been scorched (a win).

Teams are free to incorporate any sounds, visual effects, animations, or behaviors they wish, so long as the final product adheres to the properties given in their sprite description.

A break is scheduled at this point for the benefit of both the students and the organizers. While the students get up and relax, the module leaders can collect the individual sprites and organize them into a central directory structure, with one subdirectory per sprite type. With 5 different sprite types, it is helpful to have at least 10 teams, that way there are multiple solutions for each assigned type. In our case, we had a group of 30 girls and used pair programming, meaning there were three different solutions generated for each type. In addition, we added our own solution for each type within the appropriate subdirectory, for a total of four.

After the break, a short (10 minute) lecture presents the concept of composition. The recipe analogy is again useful in illustrating how the final dish (output) of one recipe can be consumed as an ingredient (input) of another recipe. In this way, several small recipes can be assembled to produce a fairly complicated and impressive final dish.

In the second part of this lab, students are asked to mix-and-match solutions for each type of sprite and assemble them into a new Scratch project. By stitching together some of the behavior (e.g., the control of dragon movement and breathing fire), and assuming that each sprite was correctly implemented individually, the result is an interactive game. Teams are then given free reign in the remaining time to extend, modify, improve, or personalize their projects in any way. A screen shot of one assembly of these sprites is show in Figure 2.

In order for the assembly of individual sprites to go smoothly, it is important to carefully and precisely define the points at which they interact. For example, a specific shade of red is part of the flame image when a dragon breathes fire. The scorched butterfly is required to burn when it touches this exact same shade of red. Similarly, the exact name is given for the message broadcast when a butterfly is scorched. This same message name is received by the score-keeping sprite.

The final code produced by each group in our recent offering of this module is available at
`http://www.cse.ohio-state.edu/~paolo/outreach/FESC07`.

The module ends with a recapitulation of the key learning outcomes and an opportunity for group discussion.

7. EVALUATION

In July, 2007, we conducted this module with a group of thirty 8th grade girls who were attending a week-long summer camp focusing on science and engineering. After completing the module, participants anonymously completed individual surveys reporting their impressions of the module

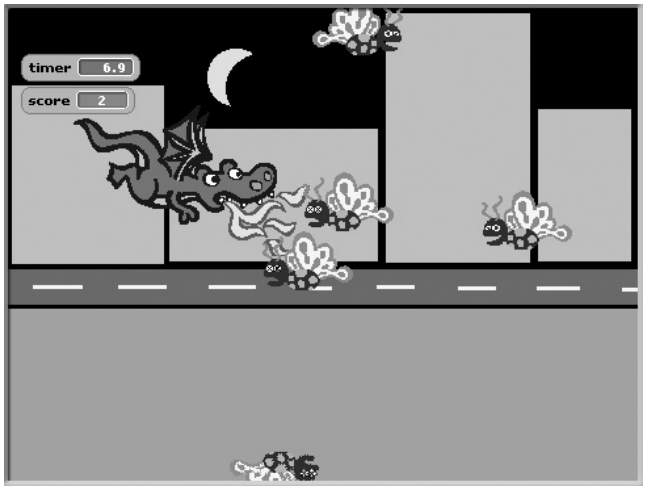

Figure 2: Sample "Dragons and Butterflies" Project

and of computer science in general. As a control, a week later we conducted a similar workshop module with a different set of 14 middle school girls. For the control group we presented only the Scratch programming environment, without broaching more advanced topics in software engineering. The control group then completed the same survey.

The survey asked students to report their prior programming experience, whether they have a computer at home, how much fun they had during the module, how much they felt they learned, how likely they were to download and use Scratch at home, and how their impression of computer science as a discipline changed as a result of the module if at all. The results are summarized in Table 1.

In terms of background, almost all students reported having a computer at home with an internet connection (27/28 in the test group and 13/14 in the control). On the other hand, few students had any prior programming experience (9/28 in the test group and 1/14 in the control). Of the 10 students with prior experience, two had used Scratch while the rest had used Lego Mindstorms or Logo.

In response to the module itself, students evaluated how much they felt they had learned and how much fun they had. Response was on a 4-point scale, with 4 being the most and 1 being the least. Both groups had similar scores in both categories, although the test group reported learning slightly more (mean 3.0 and $\sigma = 0.7$ vs. mean 2.7 and $\sigma = 0.8$). The test group also reported having slightly less fun (mean 3.2 and $\sigma = 0.8$ vs. mean 3.4 and $\sigma = 0.8$).

Beyond determining whether the module was fun for students, we were particularly curious about the likelihood of continued engagement with Scratch programming after the workshop. Students who had a computer at home were asked how likely they were to download and install Scratch within the next week. Response was on a 4-point scale, with 4 being the most likely to do so and 1 being the least. Interestingly, the test group reported a markedly higher likelihood of downloading Scratch (mean 2.9 $\sigma = 0.9$ vs mean 2.2 $\sigma = 0.7$ for the control).

Perhaps the most impressive result was the impact the use of Scratch had on students' perception of computer science as a discipline or as a career option. Many students reported changing their opinion (either "a lot" or "a little") as a result

Table 1: Summary of Survey Data

	Response	Workshop n=29	Control n=14
1. Before this lab, had you ever written a computer program?	Yes	9	1
	No	19	13
2. Do you have a computer and Internet connection at home?	Yes	27	13
	No	1	1
3. How likely are you to download Scratch at home in the next week?	Certain	9	0
	Very likely	10	4
	Somewhat likely	7	7
	Unlikely	3	3
4. How much fun did you have in this activity?	A ton	13	8
	A lot	9	4
	Some	7	2
	None	0	0
5. How much do you feel you learned in this activity?	A ton	7	2
	A lot	15	7
	Some	6	4
	None	1	1
6. Did this lab change your opinion of computer science (as a discipline or a career option) in any way?	Yes, a lot	3	4
	Yes, a little	21	9
	No, not at all	5	1
7. If you answered "yes" above, how did your opinion change? Do you now view computer science more or less favorably?	More favorably	24	12
	Less favorably	0	1

of the activity: 24/28 in the test group and 13/14 in the control group. Furthermore, of those whose opinion changed, practically *all* reported viewing computer science more favorably afterward (24/24 in the test group and 12/13 in the control group). These numbers are a testament not to the success of this module, but to the effectiveness of Scratch in engaging an audience of (female) middle school students.

Some differences do exist between our test group and the control group. Firstly, the test group was significantly larger (30 vs 14). Secondly, the test group had a narrower range of ages: All students were entering the 8th grade as compared to the control group where students were entering the 7th, 8th, or 9th grade. Finally, the two groups were taken from different summer day camps, with different administrators and hence different dynamics amongst participants. It should be noted, though, that both camps are similar in philosophy and mission. Both are zero-cost to participants, reducing the economic burden involved in enrollment. Both target underrepresented minorities and aim to select students with an aptitude in math and science, but not necessarily a long-term career ambition in engineering. Both are structured as week-long surveys of a variety of science and engineering disciplines. Thus, in our view, the control group serves as a reasonable basis for comparison.

8. CONCLUSIONS

Introducing computer science to middle school students by focusing on a programming language has the advantage of enabling continued engagement. That is, interested students can continue to program on their own, thus gaining experience in computational thinking. A different strategy for introducing computer science is to focus on an advanced topic. The advantage of this approach is that students can glimpse some intellectual underpinnings of the discipline.

We developed a 3-hour module to acheive both goals. The module includes lectures and hands-on programming labs using Scratch. It introduces several advanced concepts in software engineering, in particular: specifications, refinement, and composition. This module has been successfully piloted and found to engage students both in programming and in the advanced topics.

All module resources (lecture power points, student instruction sheets for labs, and Scratch code) are available at `http://www.cse.ohio-state.edu/~paolo/outreach`.

9. REFERENCES

[1] J. Davis and S. A. Rebelsky. Food-first computer science: starting the first course right with PB&J. In *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 372–376, New York, NY, USA, 2007. ACM Press.

[2] M. D. Ernst and J. Chapin. The groupthink specification exercise. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 617–618, New York, NY, USA, 2005. ACM Press.

[3] M. Fellows, T. Bell, and I. Witten. Computer science unplugged. http://csunplugged.org, 2002.

[4] Paolo A. G. Sivilotti and Murat Demirbas. Introducing middle school girls to fault tolerant computing. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 327–331, New York, NY, USA, 2003. ACM Press.

[5] D. J. Malan and H. H. Leitner. Scratch for budding computer scientists. In *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 223–227, New York, NY, USA, 2007. ACM Press.

[6] B. Moskal, D. Lurie, and S. Cooper. Evaluating the effectiveness of a new instructional approach. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 75–79, New York, NY, USA, 2004. ACM Press.