Experience Report: Getting Novice Programmers to THINK about Improving their Software Development Process

Tamara Caruso, Natalie Hill, Tammy VanDeGrift
Education, EECS
University of Portland
Portland, OR 97203
+1 503 943-7256
{caruso11, hilln11, vandegri}@up.edu

Beth Simon
Computer Science and Engr. Dept.
University of California, San Diego
La Jolla, CA 92093
+1 858 534-5419
bsimon@cs.ucsd.edu

ABSTRACT
Expertise is developed through both a) self-reflection and b) making useful plans for improvement [3, 10]. Traditional novice-level programming assignments require neither of these skills to be used. Could we get students to think about improving their software development processes? What areas would they identify as needing improvement? Could they write effective plans for themselves? In this experience report, we analyze the results of an intervention with 236 CS1.5 students asking them to do these activities. We find that they most commonly make improvements in planning, compared to coding and testing. Additionally, over half of the plans they make are so vague as to be of little use in helping students identify if they have, in fact, improved. Finally, we asked students at the end of the term to reflect on how their experiences with programming assignments changed over the term. We discuss our results in light of how instructors can focus instruction to help students become more meta-cognitive about their own software development processes.

Categories and Subject Descriptors
K.3.2 [Computer and Information Science Education]

General Terms
Design

Keywords
Novice, programming, software quality, metacognition.

1. INTRODUCTION
Students’ struggles with programming assignments are well documented [8]. However, in addition to programming-related cognitive or conceptual challenges, students working on programming assignments in the context of a university course face additional burdens. Kinnunen et al. [6] have reported on the emotional toll of CS1 students when doing programming assignments. They find external factors which distinguish “just programming” from “programming assignments”. These may stem, for example, from factors external to programming concepts or constructs including course-specific issues (e.g., deadlines, course load) or personal maturity (problems with time-management).

After CS1, as students begin to work on more realistic problems, their experiences with programming assignments begin to shift from primarily “programming” concerns to incorporate at least some “software development” concerns. It is the students’ experiences with, beliefs about, and self-reflection on these software development issues that this paper explores.

This report analyzes students who in their second 10 weeks of programming instruction were asked to

- Identify improvements needed in their software development skills. This was instantiated by asking students to “Write down at least one SPECIFIC plan that you can implement to improve your software development process in <the next programming assignment>”. Students were encouraged to write down more than one plan, and were prompted to provide improvement plans related to different parts of software development (planning, coding, and testing).
- Reflect at the end of the term by telling “the story of how your “experience” doing programming assignments changed from the beginning of the term to now?” Students were prompted to think of what they did during the first assignment, compared to at the end of the term.

These questions were asked as part of a larger intervention in a CS1.5 (second ten weeks of Java) course at a large research-intensive university in the western US. Overall, the instructor was concerned by the number of students who experienced extreme distress in completing the first 2-week programming assignment. For many, the challenges of 2-week long assignments seemed to linger during the entire term. Previously, exhortations made regarding the need to start early, to “sleep on” problems, and specifically to NOT start on the day the assignment was due seemed to have little impact. Even providing actual words of advice from previous students seemed to make little difference. We believed that, perhaps, some students did not think about their programming assignment experiences – at the process level. And even if they did, it was unclear if they reflected on their process in a manner which could help them improve.

Schön reports that experts in their craft reflect upon their process to assist in solving new problems [10]. Ericsson has shown that expertise is developed through the use of deliberate practice [1, 3]. That is, the transformation from novice to expert is enabled by reflection on previous success (or, usually, failure), identification of specific issues for improvement, and then followed by deliberate and conscious practice where those issues are addressed. Examples abound in the arena of sports. Top athletes engage in continuous, highly-scrutinized training plans where specific performance issues are targeted for improvement. Our current programming assignments did nothing to support expertise development in this...
way. Could we help students move beyond “getting a program to work” to identifying what they could do to “improve their game”?

Our approach was lightweight in structure – asking students to complete a “software development” reflection through a series of open-ended and multiple choice questions, delivered via an online survey after most assignments. The multiple choice questions asked students to identify if they had made any improvements in a given process area (planning, coding, and testing) after their second programming assignment. Students reported making more improvements in the area of planning than coding or testing. When asked to develop up to 6 specific plans to use in improving their software development processes for the next assignment, students produced an average of 4.1 per student, with many of them referring to the planning stage and coding activities. Most of these plans were implementable activities, but stated so vaguely as to be challenging to assess. This is an important component of deliberate practice. If a person sets an improvement goal which cannot be objectively evaluated, it is unlikely to be effective in helping him/her make an improvement (e.g., “I’ll get in shape this fall” versus “I’ll go to the gym T/Th and do 30 minutes on the treadmill”). Finally, at the end of the term, students were asked to describe in their own words the transformation of their software development experiences as applied to programming assignments across the 10 weeks of the course. Just 5% of students reported transformations in testing and 20% in coding. More transformations came in the planning stage (40%) and better time management (32%).

2. RELATED WORK

Though this work draws primarily on general research on development of expertise [3, 10], there has also been work looking at students’ reflections in introductory computing. In [4] they describe different techniques to encourage reflection in computing courses. They stated that “evidence suggests that reflection actually enhances the technical master achieved”. Hanks et al. [5] also gathered students’ reflections by asking them to provide advice to fellow students in introductory programming courses. They found categories of general, programming-specific, and attitudinal advice – which was impacted by an intervention to get students to adopt a growth-mindset. Our approach is more specifically targeted on the software development process -- requiring students to submit bi-weekly planning sheets and diaries that reflect on their previous programming assignment experiences and reflect on specific, measurable plans to improve.

Although the core of this paper reports on students’ development of meta-cognitive skills, this is studied within the context of novices’ software development processes. Others have reported in this area. Edwards et al. [2] show results from 5 years of data that, compared to their classmates, A and B earning students start and finish their assignments earlier, wrote more code, but did not necessarily spend more time. Kolikant [7] examined students’ definition of correctness especially as it relates to testing. She found that students believe an adequate testing strategy includes testing all the examples he or she can think of – a strategy instructors find incomplete. Porter and Calder [9] describe a study exploring the use of patterns in beginning programming classes found that requiring students to plan a solution before coding was important in getting students to use patterns.

3. METHODS

3.1 Course and Student Information

The data for this study was collected at a large public research-intensive university located on the west coast. The study took place in a 10-week undergraduate CS 1.5 course covering Java and object-oriented programming. Data was collected in two sections of the course (Winter 2010 and Spring 2010) with 163 students enrolled in winter and 73 students enrolled in spring. Eighty percent of the students in winter were CS majors; however, only 45% of the students in spring were CS majors. The course requires 5 two-week long programming assignments, and follows a CS1 course which uses pair programming for 9 1-week long programming assignments.

3.2 Data Collection

Students completed homework (via online surveys) that asked about creating plans for creating their programming assignments, students’ perceptions of things they could improve on, the effectiveness of improvement plans, and a reflection on their development as programmers during the term. Slightly varying polls were given in Winter and Spring, but we report here only on questions identical in both terms. Questions varied: some were closed-form (check boxes, radio buttons); others were open-ended questions (text boxes). Students had the option to not respond to some questions, and not all enrolled students completed every survey. Table 1 summarizes the data collection.

Table 1: Number of respondents to each survey

<table>
<thead>
<tr>
<th>Survey ID (given after which assignment of 5)</th>
<th>Number of Responses in Winter and Spring</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1: Survey 1 (after 2nd)</td>
<td>W=109 S=64</td>
</tr>
<tr>
<td>S2: Survey 2 (after 3rd)</td>
<td>W=111 S=67</td>
</tr>
<tr>
<td>PT: Post Term Survey</td>
<td>W=113 S=73</td>
</tr>
</tbody>
</table>

3.3 Data Analysis

The answers to each question were analyzed independently. Closed-form and open-ended questions were analyzed differently, as described below.

Closed-form questions: students chose from a set of pre-formed answers that best fit their experience and the chosen answers were tallied. Examples include whether they thought a plan was effective or not, how useful a plan was, or how long they spent on a programming exercise.

Open-ended questions: students responded to prompts which asked for their advice, their strategies, what worked for them, and their stories of development over the term. Three authors analyzed the open-ended responses and classified them into emergent categories, with some responses fitting into more than one category. Then the broad categories were condensed and the answers were analyzed further.

Though a number of other related questions were asked in these surveys, this paper only reports on the subset which informs the following research questions:

- How did students believe they improved their software development process?
- How many and what kind of improvement plans did students create?
• What self-identified transformations did beginning programmers report occurred over the term?

4. RESULTS

4.1 How Did They Believe They Improved?
Students were asked to respond to questions about strategies they used to improve their programming experience in the second assignment (S1). This was done by asking separately about the tasks of planning, coding, and testing. For each task students completed one “select all that apply” question providing a list of things they might have improved upon. The first option on each list was “I didn’t improve on any <task> activities.” The options presented in each process stage were generated by the instructor based on anecdotal evidence regarding students’ struggles with the assignments. Immediately following, students were asked in an open-ended question to give a specific description of what they did to make that improvement. This allows us to better understand how students interpreted the available options in the first question.

4.1.1 Planning
The select-all question asked students to check if they improved any of the following when planning. The numbers for each improvement appear in the table below.

<table>
<thead>
<tr>
<th>Improvement in planning</th>
<th>Number (Winter, Spring)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Practicing/understanding fundamentals before starting</td>
<td>37 (33.9%), 29 (45.3%)</td>
</tr>
<tr>
<td>Reading the textbook more</td>
<td>34 (31.2%), 13 (20.3%)</td>
</tr>
<tr>
<td>Understanding the specification</td>
<td>63 (57.8%), 27 (42.2%)</td>
</tr>
<tr>
<td>Planning a solution</td>
<td>41 (37.6%), 13 (20.3%)</td>
</tr>
<tr>
<td>Other (most were started earlier)</td>
<td>10 (9.2%), 0 (0.0%)</td>
</tr>
<tr>
<td>I didn’t do any better planning activities</td>
<td>27 (24.8%), 19 (29.7%)</td>
</tr>
</tbody>
</table>

The average number of improvements reported by students, including those that reported no improvement, was 1.69 in winter and 1.29 in spring (out of 4). There were 46 students over both terms who selected that they did not do any better at planning on the second assignment.

The most frequent open-ended responses about recommendations to improve planning were:
1. Read specification carefully (28 winter, 27 spring)
2. Start earlier (28 winter, 11 spring)
3. Read textbook (22 winter, 13 spring)
4. Make outline, plan solution (14 winter, 13 spring)

Students talking about understanding the specification almost always indicated that they would recommend reading the specification more carefully. For example, one student commented that he/she would improve planning by: “Read the specification very carefully and know what you're ultimately trying to do with a method when you write it.”

4.1.2 Coding
Students answered identical closed-form and open-ended questions about their coding tasks. The average number of improvements reported per student, including those who did not have improvements, was 2.42 in winter and 1.62 in spring (out of 6). There were 28 students who reported making no improvements in coding.

<table>
<thead>
<tr>
<th>Improvement in coding</th>
<th>Number (Winter, Spring)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I organized my code better</td>
<td>52 (47.8%), 20 (31.3%)</td>
</tr>
<tr>
<td>I compiled more often</td>
<td>74 (67.9%), 26 (40.6%)</td>
</tr>
<tr>
<td>I was better at compile-time debugging</td>
<td>47 (43.1%), 19 (29.7%)</td>
</tr>
<tr>
<td>I was better at run-time debugging</td>
<td>30 (27.5%), 11 (17.2%)</td>
</tr>
<tr>
<td>I ran into fewer roadblocks</td>
<td>25 (22.9%), 12 (18.8%)</td>
</tr>
<tr>
<td>I asked for help more effectively</td>
<td>33 (30.3%), 15 (23.4%)</td>
</tr>
<tr>
<td>Other</td>
<td>3 (2.8%), 1 (1.6%)</td>
</tr>
<tr>
<td>I didn’t do any better on coding activities</td>
<td>10 (9.2%), 18 (28.1%)</td>
</tr>
</tbody>
</table>

The top open-ended responses about recommendations to improve coding were:
1. Compile often (34 winter, 11 spring)
2. Plan before writing code (15 winter, 8 spring)
3. Ask for help from tutors and TAs (16 winter, 2 spring)

Here are some example recommendations made by students from each of three categories above:
1. “compiling more often can help your debugging as it allows you to see your mistake as you make it”
2. “i commented on each method before doing anything else, then after all that i started programming.”
3. “When you are stuck in not knowing how to code something, ask a TA for help, but make sure you know exactly what you wanna [sic] do.”

4.1.3 Testing
Students were also asked what testing strategies they found effective for producing quality code. The average number of improvements reported per student was .72 in winter and .67 in spring. There were 89 students who reported making no improvements in testing.

<table>
<thead>
<tr>
<th>Improvement in testing</th>
<th>Number (Winter, Spring)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I brainstormed ahead of time possible erroneous situations</td>
<td>33 (30.3%), 16 (25.0%)</td>
</tr>
<tr>
<td>I developed test cases</td>
<td>27 (24.8%), 13 (20.3%)</td>
</tr>
<tr>
<td>I more robustly tested my code</td>
<td>17 (15.6%), 14 (21.9%)</td>
</tr>
<tr>
<td>Other</td>
<td>1 (0.9%), 0 (0.0%)</td>
</tr>
<tr>
<td>I didn’t do any better on testing activities</td>
<td>55 (50.5%), 34 (53.1%)</td>
</tr>
</tbody>
</table>
Here are the top open-ended student-generated strategies for
improving testing:

1. Test each method/class individually (9 winter, 12
   spring)
2. Predict errors and try to make errors/break program (10
   winter, 9 spring)
3. Create test cases (16 winter, 2 spring)
4. Do nothing different (11 winter, 0 spring)
5. Test early (0 winter, 8 spring)

Here are some examples of students’ specific strategies for
improving testing:

- “I created a test main method to use on all my methods
to make sure they worked properly and implemented what I
wanted them to do”
- “Try finding ways to make your code fail and then fix those
holes.”
- “I made a test sample of the data and practiced my methods
on that sample in the interactions panel”

4.2 Plans for Improvement

After being asked to reflect on things they had done, students
were encouraged to develop plans for what they could improve
for the next assignment (3rd). They were asked:

Write down at least one SPECIFIC plan that you can implement
to improve your software development process in <the next
assignment>. The more plans you make and the more specific
you are, the more likely they will help you.

They were provided with multiple textboxes to fill in with plans:
2 for planning tasks, 3 for coding, 1 for testing, and an “Other”
plan. Students were required to fill in at least one plan to
complete the survey.

The average student identified 1.75/1.69 improvement plans for
the planning stage, 1.65/1.52 for the coding stage, and .70/.66 for
the testing stage. The grand majority of plans fell in the planning
and coding stages.

Table 5: Student Planning Goals.

<table>
<thead>
<tr>
<th></th>
<th>Planning</th>
<th>Coding</th>
<th>Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Percent of students creating at least one plan for this stage</td>
<td>95% W, 92% S</td>
<td>96% W, 67% S</td>
<td>62% W, 59% S</td>
</tr>
<tr>
<td>Percent of those plans “clearly quantifiable”</td>
<td>45% W, 35% S</td>
<td>45% W, 42% S</td>
<td>62% W, 49% S</td>
</tr>
</tbody>
</table>

But were these goals actually useful to students? A primary goal
of this activity was to help students focus on deliberate practice –
part of which involves setting a clear, measurable goal for
themselves. If you have a vague goal it is hard to say whether
you met it, which leads to challenges in identifying whether any
change you observed in your performance was attributable to that
change in plan. Before being asked to write their plans, students
were shown this motivation, which is based off the discussion in
[1]:

As every good sports coach will tell you, people with non-specific plans for “improving their game” don’t do very well in actually improving. Kobe Bryant doesn’t just go out and shoot baskets each day. Serena Williams doesn’t “just” hit balls. They have a specific training regimen and identify specific measurable goals for improvement and track their progress in attaining them. Learn from Kobe and Serena -- improve your software development processes.

Students’ plans were coded as being either “clearly quantifiable activities,” or “vague activities” – which was anything else.

Examples of clearly quantifiable activities include:

- Read the Spec and highlight important key items (planning)
- Compile at each minute (coding)
- Write out a rough framework of the code on paper then attempt to transfer this framework to the compiler with comments about what and how I intend to code it. (coding)
- Test each method as I write them (testing)
- Apply system.out to make sure that everyone works as intended (testing)

Examples of “vague” activities include those that clearly indicate doing something (differently) but are vague enough that one can imagine a student, after the fact, not being able to prove that they had been successful at implementing this plan

- Spend more time brainstorming possible coding to use for certain tasks in the PSA (It’s unlikely the student knows how much time they spent previously).
- Read the spec farther in advance to have more time to think about it (What day will you have read it by?)
- Map out a course of action before you start coding (What is in this map? Which calls what? The purpose of each method in English?)
- Use the interactions pane more to test whether a function works. (Close maybe, can you describe how?)
- Write main as soon as possible to make sure methods work (How soon is that? Can’t you do it first?)
- Comment the codes before i move on (quite close, indicate level of granularity of commenting)
- Create more effective, less time consuming test cases/methods (what is the metric for effective? Time consuming?)
- Start early so i have time to test it out (what day?)

Words that frequently accompanied “vague” plans include: understand, think about, plan out, organize, check, and make sure. Comparators also were seen in vague plans: more, earlier, and often among them. These are all terms one can imagine a student being able to “fudge” about, when reviewing their plan after the fact.

It should be noted that specificity alone does not determine the potential usefulness of a plan in improving students’ experiences. For example, “Read the spec before beginning coding” is clearly quantifiable – a student can answer “yes I did that” or “no I did not do that”. However, an instructor can use such an example to help students develop expertise on the content of plans which are possibly more helpful. But getting students to be able to write specific plans is a critical, hopefully easier, first step.

4.3 Transformations Over the Term

At the end of the term (after the last assignment was due) students were asked the following:
Can you tell us the story of how your “experience” doing programming assignments changed from the beginning of the term to now? You might want to describe your typical approach to a programming assignment back in week 1 and now.

These stories were coded into emergent categories. Some stories contained more than one change, so each change was coded separately. For example, a student may have stated that she started assignments earlier and read the specification more carefully – which was coded as two separate transformations. In total, there were 143 transformations in Winter (N = 113) and 98 separate transformations in Spring (N = 73). Table 6 lists the transformations and the aggregate number of students who reported a transformation in that category.

Table 6: Self-reported transformations for students in both Winter and Spring terms

<table>
<thead>
<tr>
<th>Transformation</th>
<th># of students (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Better time management</td>
<td>60 (32.2%)</td>
</tr>
<tr>
<td>Planning</td>
<td>43 (23.1%)</td>
</tr>
<tr>
<td>Coding and debugging</td>
<td>37 (19.9%)</td>
</tr>
<tr>
<td>Using help resources (tutor, online)</td>
<td>35 (18.8%)</td>
</tr>
<tr>
<td>Reading spec more carefully</td>
<td>34 (18.3%)</td>
</tr>
<tr>
<td>Testing</td>
<td>10 (5.4%)</td>
</tr>
<tr>
<td>Internal change (confidence, indep.)</td>
<td>9 (4.8%)</td>
</tr>
<tr>
<td>Other</td>
<td>6 (3.2%)</td>
</tr>
<tr>
<td>No change</td>
<td>7 (3.8%)</td>
</tr>
</tbody>
</table>

One can see that, collectively, students made transformations throughout the software development process, from the planning stages through the testing stages. Not surprisingly, as the assignments became more difficult, almost a third of the students felt the need to start assignments earlier and become better at managing their time. Several students also realized the importance of planning and documented this change in behavior in their stories.

“In the beginning of this term, I wasn’t so good at managing my time. Last term, my programming partner always set up times to meet up to work on the PSAs. However, this term, being on my own, I would often leave much of the work to be done on the day that the PSA was due. And, this was also a bad habit because the PSAs this term were more complex, not to mention longer and more in depth, especially since we only did 5 this term. After making the mistake of procrastinating on the first two PSAs, I eventually learned to make sure to set aside adequate time to code, debug, test, etc.”

Some students also noted the importance of carefully reading the entire specification.

However, as the quarter went by, I realized that I should read the entire specifications before I dive into the PSA. This helped a lot in planning what I was going to do before I started to write the code.

The strategies offered regarding coding and debugging included frequent compilation, using print statements, and commenting first.

A big change that occurred throughout the quarter was when I commented my code. In the beginning I would wait to comment until the very end but now I start commenting in the beginning before I write my code. This helps me maintain an idea of how I should approach the assignment. Also, whenever I came upon a problem like my code not compiling or getting a run time error I would usually depend on asking a tutor but then I started to tracing my code and using print statements to figure out the problem myself. This is a valuable skill that I began to develop. Lastly, after I finished my PSA I would return and review how I approached the assignment in order to have a better understanding of the concepts.

A few students noted more internal transformations, reporting that they had more confidence in programming and they felt more independent.

“I feel like I have improved significantly in terms of how independently I program. I felt like I really needed the tutor’s help at the beginning and did not understand how crucial it was to start early. For my last PSA I started the day after it was posted and did it completely by myself.

The most significant change has been my confidence in creating programs with little instruction. At the time I am writing this, I feel that I have done so many programming assignments, and usually done so much extra work on them, that all the elements of programming we learned in this class are fixed securely in my mental toolbox.

5. DISCUSSION

Overall, students in CS1.5 seem highly focused on improving their software development skills in the area of planning (compared to coding or testing). A high percentage of plans for improvement they make are in the planning arena. However, these are also “easy” areas to describe improvements in – involving starting earlier, reading more carefully, etc. Developing specific plans regarding the coding process may be more intimidating as they have less experience with and possibly “control” during that part of the programming process. Perhaps instructors should provide more specific advice or scenarios for students to follow in building better coding skills. Compile after every line or two – perhaps this is something students can grok. Are there better “plans” we can help them adopt?

What about the original motivation of trying to exhort students to pay more attention to process, or at least to start early on their assignments? While students may not take this advice at the beginning of the term, their reflections on their growth show that...
they have gotten the picture loud and clear by the end, with a third of students identifying this as something that changed in their programming “experience”. Maybe this is simply a growth process that students must experience for themselves. Some can even tell us poignantly what they know they must do:

With the first assignment, I waited until nearly the last minute, figuring it would be easy, since it was the first assignment. I was wrong, and the extension offered until Sunday saved me, even if my grade for the assignment took a hit. Since then, I vowed not to save my assignments for the last minute, let alone the second week. So assignment two came along, and though I promised myself I would get started on it when I was supposed to, the last minute came, and I only finished about half of the assignment. I again promised myself to start on time for the next one. And though I completed assignment three on time, the same couldn’t be said for four and five. I don’t have an excuse for why I didn’t do the work, and I can’t really say I’ve learned, at least in practice, how to approach the assignments. I know what must be done, I just didn’t do it. All I can say I learned is that I need to change all of my habits if I’m going to see any improvement in the future.

Which hints at the last concern -- that students’ abilities to develop appropriate, truly measurable plans for improvements are notably lacking. They may have the right ideas, but could use greater direction in incorporating that key specificity of “what I will do differently” that has been shown to be necessary for engendering deliberate practice. A series of examples may be enough, but likely we need to require our students to incorporate effective reflective practice activities as part and parcel of their programming assignments. We are not just trying to teach them “Java” – and we will not always be there to help them further hone their software development skills. We need to instill the meta-cognitive skills that will allow students to learn new languages and continue the development of their expertise throughout their professional careers. We encourage future work examining more specifically the deliberate practices we should be coaching our students in applying.

6. CONCLUSION
This study used survey instruments to encourage self-reflection skills regarding students’ software development processes. Self-reflection is an important component in metacognition, a practice experts use when solving problems [10]. Most introductory computing courses teach the fundamentals of a programming language and rely on teaching software process in a later course. However, we took the approach of incorporating reflection on process during the second ten weeks of learning programming. Even though we targeted a CS1.5 course, these reflection activities could easily be incorporated into the latter half of a CS1 course. Our results showed that most students self-identify improvements in planning, coding, and testing; however most of their improvements come in the planning and coding stages. Over half reported no improvement in testing practices. Perhaps CS1.5 students do not have enough experience to focus improvements in the testing stage or they do not spend much time on this stage; this result may indicate that students and teachers value planning/coding activities more than testing. Most students could create improvement plans; however, over half of these plans were too vague to be assessable quantitatively. Although not reported on here, most of the students who remembered their plans from the previous assignment stated the plans were effective for improving their process and easing frustration on the next assignment. Other instructors may want to incorporate deliberate planning exercises in their introductory courses to encourage reflective behavior. Novice students need scaffolding not only with learning how to program, but also in developing planning and reflection skills.

ACKNOWLEDGMENTS
Our thanks to the students enrolled in the CS1.5 course for generously providing survey responses for data analysis. We also thank Dr. James Male for providing funding for undergraduate students to engage in this research project.

7. REFERENCES