Work In Progress - Using a Visual Programming Language to Bridge the Cognitive Gap Between a Novice's Mental Model and Program Code

B.J. Smith, Harry S. Delugach
University of Alabama Huntsville, smithbj@uah.edu, delugach@cs.uah.edu

Abstract - Current research suggests that many students do not know how to program at the conclusion of their introductory course, which has been taught predominately with textual and auditory lecturing. By primarily appealing to programming novices who prefer to understand visually, an understanding method not currently accommodated through the standard lecture style used in most classes, we develop a method that encourages communication of programming solutions. This method builds upon previous research that suggests that most engineering students are visual learners and we contribute that using a flow-model visual programming language will address important and difficult topics to novices of programming. We performed a pilot study using a knowledge modeling tool instead of using an existing visual programming tool to test this method, and share the program understanding results using this theory.

Index Terms – Index of Learning Styles, RAPTOR, Flow-Model Tools, novice programmer

INTRODUCTION

Programming is hard, and can account for a portion of the reason that computer science has a 30-50% attrition rate [1]. Current research also suggests that many students do not know how to program at the conclusion of their introductory course [2] which has been attributed to novices not forming valid mental models. Felder laments that “Unfortunately, a single approach has dominated engineering education since its inception: the professor lectures and the students attempt to absorb the lecture content and reproduce it in examinations. That particular size fits almost nobody: it violates virtually every principle of effective instruction established by modern cognitive science and educational psychology [3]-[4].” [5] The field of computer science is losing valuable minds to the field just because novices cannot program, get frustrated, and do not pursue the science further. In this work we develop a method that encourages communication of programming solutions by primarily appealing to programming novices who prefer to understand visually (per the Index of Learning Styles (ILS) Questionnaire [6]), an understanding method not currently accommodated through the standard lecture style used in most classes [7]. We argue that the lecture method made sense when the printing press was invented, and text was much easier to mass produce than images, but that visualization is easier to reproduce now, and we should consider updating our system.

RELATED WORK

In an introductory IT course at the U.S. Military Academy, a customized version of Rapid Application ProtoTyping of Objects and their Relations (RAPTOR) was compared to an existing commercial software visualization package, Microsoft Visio [8]. The experiment [9] focused on three questions (that were chosen out of the entire exam). The results of the experiment showed that the class that used RAPTOR answered those three questions more accurately than those classes that used Ada or MATLAB. This is even more remarkable given that the RAPTOR class had average GPAs entering this class lower than the average GPAs of the students entering the other classes (2.47 compared to 2.84). Since the RAPTOR class only had 16 students in it, the sample size was too small to achieve statistical significance, even though the other classes had 365 students.

With these encouraging results, their department decided to replace the existing software visualization tool with RAPTOR in all sections, beginning in Fall of 2006. The results are detailed in [9]. Subsequent statistics of the class were collected with statistically significant sample sizes, which can be found in [10]. Their work did not account specifically for learning styles.

LEARNING STYLES

Several dozen different learning style models have been developed. The best known of these models is Jung’s Theory of Psychological Type [11] as operationalized by the Myers-Briggs Type Indicator (MBTI) [12]. Other models that have been applied extensively to engineering are those of the Kolb Cycle by Kolb [13] and the Index of Learning Styles [9] by Felder and Silverman [14]. Even as the concept of learning styles is not universally accepted [5], we have chosen to use Felder’s ILS as our basis based on the analysis on reliability and validity done by Olds et al. in [15]. Table I describes and contrasts the different dimensions in the ILS.
Table I

THE FOUR DIMENSIONS OF THE ILS

| Sensing - concrete thinker, practical, oriented toward facts and procedures | Intuitive - abstract thinker, innovative, oriented toward theories and underlying meanings |
| Active – learn by trying things out, enjoy working in groups | Reflective – learn by thinking things through, prefer working alone or with a single familiar partner |
| Visual – prefer visual representations of presented material, such as pictures, diagrams and flow charts | Verbal – prefer written and spoken explanations |
| Sequential – linear thinking process, learn in small incremental steps | Global – holistic thinking process, learn in large leaps |

Table III

FINAL PROGRAMMING GRADE AND FINAL OVERALL GRADE RESULTS

<table>
<thead>
<tr>
<th></th>
<th>Test Group</th>
<th>Control Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Final Programming Grade</td>
<td>96 ± 4</td>
<td>94 ± 8</td>
</tr>
<tr>
<td>Final Overall Grade</td>
<td>94 ± 8</td>
<td>92 ± 8</td>
</tr>
</tbody>
</table>

Figure I

THE TEST GROUP'S LEARNING STYLE DISTRIBUTION

Figure II

THE CONTROL GROUP'S LEARNING STYLE DISTRIBUTION

Our Approach

By normalizing the corresponding important and difficult computer science concepts from a survey found in [16], we create a list of computer science concepts that are relevant to apply our method to.

We would like to measure the relationship between learning style and performance of novices in an Introduction to Programming course. We performed a pilot study using slides that combine both text and flow-model diagrams in an introductory computer science course. The slides described homework, class assignments, and programs for a part of seven lectures. This pilot study returned encouraging, yet statistically insignificant results since there were only 11 students that participated in each group, summarized in Table III. Figure I show the Test Group’s Learning Style distribution, and Figure II shows the Control Group’s Learning Style distribution.

Table III

FINAL PROGRAMMING GRADE AND FINAL OVERALL GRADE RESULTS

<table>
<thead>
<tr>
<th></th>
<th>Test Group</th>
<th>Control Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Final Programming Grade</td>
<td>96 ± 4</td>
<td>94 ± 8</td>
</tr>
<tr>
<td>Final Overall Grade</td>
<td>94 ± 8</td>
<td>92 ± 8</td>
</tr>
</tbody>
</table>

One deficiency of our pilot program was that we did not use an existing visual programming tool for our visualization. We consider this a deficiency because the instructor had to create individual slides by hand, and modify the visualization’s execution output step by step using a software program that supported flow-model visualization, but did not enforce programming language rules. This was error-prone for an experienced instructor, and would be overwhelming for the novice computer science student. In our next experiment, in which our subjects will be introductory psychology students, we will produce slides created from an existing tool, RAPTOR, to our novices at the same time that we cover new computer science topics in text.

We would like to use an existing robust visual programming tool that enforces visual syntax because in future experiments we believe that this will increase the novice’s engagement, which has been shown to improve their performance [17]. In this future curriculum, we would like to modify RAPTOR to include features that would make it a general purpose programming language, something that RAPTOR does not currently try to address. Such features would include adding a seamless translation from the diagrammatic language to an existing textual programming language, displaying the diagrammatic code concurrently with an existing textual programming language. Furthermore, the issues of scale must be addressed, and we could take cues from Code Bubbles [18] and also should consider utilizing the Map metaphor [19].

References

Session F3G


Author Information

B.J. Smith, PhD Candidate, University of Alabama Huntsville, smithbj@uah.edu

Harry S. Delugach, Director, Honors Program, Associate Professor, Advisor, University of Alabama Huntsville, delugach@cs.uah.edu