World Academy of Science, Engineering and Technology
International Journal of Educational and Pedagogical Sciences
Vol:1, No:3, 2007

Using Visual Technologies to Promote Excellence in Computer Science Education

Carol B. Collins, and M. H. N Tabrizi

***Abstract*—**The purposes of this paper are to (1) promote excellence in computer science by suggesting a cohesive innovative approach to fill well documented deficiencies in current computer science education, (2) justify (using the authors' and others anecdotal evidence from both the classroom and the real world) why this approach holds great potential to successfully eliminate the deficiencies, (3) invite other professionals to join the authors in proof of concept research. The authors' experiences, though anecdotal, strongly suggest that a new approach involving visual modeling technologies should allow computer science programs to retain a greater percentage of prospective and declared majors as students become more engaged learners, more successful problem-solvers, and better prepared as programmers. In addition, the graduates of such computer science programs will make greater contributions to the profession as skilled problem-solvers. Instead of wearily rememorizing code as they move to the next course, students will have the problem-solving skills to think and work in more sophisticated and creative ways

***Keywords*—**Algorithms, CASE, UML, Problem-solving.

I. Introduction

IN many countries including USA, the number of computer science students continues to decline. Students who show an initial interest in the field drop out in substantial numbers as the tedious realities of traditional programming instruction emerge.

A. Evidence of Deficiencies in CS Education

Most entry-level programming courses focus on coding and introduce students to Object-Oriented Programming (OOP) in C++ or JAVA [14, 21]. The following summarizes the resulting problems cited in the literature:

- Students focus on the syntax of the programming language and begin solving the problem by using the programming language, more often by trial and error, rather than first analyzing and designing solutions to the problem.
- Since students are unable to make a connection between problem-solving and coding, they often lose motivation and ultimately switch to other degree programs.
- Without a basic understanding of software design and programming concepts, those students who stay with the program face an uphill battle in dealing with more complex programming related CS courses.
- Students develop unproductive habits like mimicking code and tinkering with the code to fix problems.
- Professors must repeatedly teach the same topics because students have not learned concepts that are transferable across topics and curriculum levels.
- Students enrolled in senior level courses such as compiler construction and computer graphics spend excessive amounts of time with coding when working on projects instead of mastering the new concepts that the course and project are supposed to emphasize.
- The inability to code effectively and efficiently becomes more serious when students take the software engineering courses. With years of course work in OOP, students often cannot program adequately. For students who lack an understanding of programming concepts and problem-solving capabilities, the syntax of the programming languages simply overwhelms them.

The authors have witnessed the same symptoms in the senior level software engineering as in introductory computer science courses. Even the good programmers mimic or tinker with code, not truly understanding concepts behind programming.

Although these students already had taken several semesters of courses using OOP languages like data structures, compiler, database, and computer graphics, they could not clearly state in English what an "object" was without referencing a particular syntax of a programming language. These deficiencies often are published and discussed [9, 12, 15, 20] at computer science and information technology-related conferences.

The symptoms are not an aberration; they are the norm. To convince the students that the process itself of creating software is indeed linked to software quality, in the software engineering course, the authors reverse engineered students' cherished "A" rated programs from earlier courses. Upon seeing the result, the students were appalled at their programs' structure and design. Clearly, knowledge and proficiency in writing code is a necessary but insufficient first step to create object-oriented software [7]

Often only a few graduating seniors are viewed as good programmers. At Microsoft PDC03 conference, most of the participating Faculty and Deans from leading universities

Manuscript received March 8, 2007.

Carol Collins is with East Carolina University, Greenville, NC 27858 USA (phone: 252-328-9692; fax: 252-328-0715; e-mail: collinsc@ ecu.edu).

M. H. N. Tabrizi is with East Carolina University, Greenville, NC 27858 USA (e-mail: tabrizim@ ecu.edu).

throughout the USA, at the general academic session, felt that most of their graduates could not program effectively.

One can also cite the results of the last year's ACM World Finals Programming Contest sponsored by IBM: MIT was the highest placing university in the USA at position 8, and the next highest USA University, CIT University, placed 39. One cannot say that the USA did poorly because of financial resources devoted to a select few students, when an elite university like MIT placed eighth and no other USA university placed above 39.

B. Approaches to Address the Deficiencies

ACM: ACM's Computing Curricula 2001 is a major contribution as far as it goes. But evidently from the number of curricula revision requests from NIH, the document does not go far enough in supporting problem- solving. For example, Chapter 7 lists the deficiencies already cited and links these to the current emphasis on early coding. Three alternatives are suggested, but all suggestions involve listing and ordering topics, and do not address how to provide support for the problem-solving needed, although "developing cognitive models" is mentioned. (Chapter 7.5). Even in the table of activities, the activities imply verbal descriptions (for example, "describe" is used, not "sketch"). Even in the "Algorithms First approach", where visual modeling would seem natural, the modeling referenced is pseudo-code. Imagine a building architect describing what is in a blueprint with some sort of pseudo-code!

The next chapter deals with "Intermediate Courses"—at some point pseudo-code is abandoned, but what takes its place? Here is where the study of computer science is really fragmented, creating an impression of a hodgepodge of topics but not a unified field like physics or biology.

The authors contend that whatever the topics and whether coding is early or late, without models to support the thinking involved in the solution process the situation will not improve. In particular, visual modeling seems to offer a unifying approach to problem-solving that allows students to build upon and expand the problem-solving techniques already learned instead of abandoning them as new topics are introduced. The ACM updated Computing Curriculum report's focus is to specify curricula specific to subfields of computing and computer science, like software engineering and IT. The issue of a coherent toolbox with appropriate problem-solving tools is again addressed in passing.

Textbooks: Textbooks generally offer local remedies, but no support for problem-solving in the context of programming, e.g., popular texts like Savitch's [18]. Analysis with real data gets little attention even if textbooks note problem-solving structures involving branching, looping, and recursion. Design models often are expressed as pseudo-code or code. No wonder students think that problem-solving starts with code!

Using pseudo-code merely avoids some complexity involving syntax of programming language while offering nothing or very little in the way of guidance in the early stages of problem-solving. Other textbooks like [2] show once visual models (flow diagrams) of coding structures (e.g., if/else) but then do not use these for problem-solving. Instead, example solutions start with code or pseudo-code.

Other remedies, including class diagrams, also have been proposed and appear in newer textbooks [6]. Visual-based class diagrams represent a potential improvement over pseudo-code. However, being static design models, class diagrams are fixed and structured too close to the code level. These diagrams will not fully support the students' engagement in the problem-solving process from the beginning.

In addition, approaching software development from pseudo-code or the class diagram level requires that the student already know how the problem should be solved -- often by using step-by-step and algorithmic methodologies. Moreover, writing pseudo-code to describe what needs to be done is like describing a movie with prose. In summary, these approaches generally compress into linearity the inherent non-linearity of the solution process.

Other approaches based on software aids, (e.g., BlueJ [3]); memory diagrams [10] also require thinking that is too close to the code level. Functional programming languages, like Dr. Scheme have built-in analysis and design support via the "design template" as suggested by the author [8]. However, this template is specifically suited to functional programming and is also nearer the code level. Alice [1] is one of the better approaches that can be used to support analysis and design first, but can also be misused, allowing and even encouraging the habit of using only trial and error tinkering. Alice does seem to contribute to retention and better attitudes as measured on standard scales [13].

What is missing in all of these approaches is a unified and formalized methodology that involves a general process with effective tools that:

- Support problem-solving irrespective of the code that will ultimately be produced.
- Enable students to apply the principles and approaches to problem-solving and programming, and
- Show the interaction of these approaches, to include creating granularity, abstraction, top-down, divide-and-conquer, foot-in-door, modularity of functionality, and flow of events.

Moreover, such a unified method can be enhanced by the use of the other existing aids, including memory maps, BlueJ, Alice, and design templates, depending on the specific code to be used. In fact, these aids would seem to make more sense and to be used more effectively if presented as enhancements to a common process with toolbox instead of as isolated pieces.

Overall, the collective effort to overcome the difficulties related to syntax-based teaching of programming courses has been piecemeal. Proposed methodologies have focused on specific languages and provided solutions for specific problems in specific courses, or addressed thinking that is too close to the code level.

However, our methodology, based on the authors' extensive anecdotal experience over many years, is grounded in a

process supported by visual modeling, is broadly applicable, and will fit with current textbooks and be applicable as the breadth of computer science continues to increase. Problem-solving based in visual models, already demonstrated for the engineering fields and other sciences, can create an environment beginning with existing partially effective systems (like BlueJ and Alice) can be seen as part of a larger picture and be more effective in aiding learning throughout a student's academic and professional career in computer science.

Moreover, the student will be prepared for today's world where visual modeling is becoming ubiquitous (e.g. see VB Studio and its visual models of GoF patterns).

II. The Visual Modeling Approach

Based on their own classroom experiences over the years, the authors have seen that an effective problem-solving methodology becomes an iterative process supported by visual modeling tools. The advantage is that students have appropriate visual modeling tools that can be used throughout the computer science curriculum. Moreover, as the complexity and scope of problems increase, the process and tool set can be augmented not supplanted. The keys are "enough complexity" and "appropriate tools" at each curriculum level. In this way, students get to practice thinking that is repeated within CS1 and throughout the curriculum, just as the physical science students do using their methods with visual models.

Two major consequences can occur by using this approach: (1) Professors need not spend lots of time re-teaching because the concepts were presented too close to the code level. (If concepts are taught too close to the code level, students associate the concepts with the code; thus cannot apply the concepts when the programming language changes.) (2) The approach naturally evolves into processes widely used by professionals, like the Rational Unified Process (RUP) supported by UML and Rational Rose [17].

A. Visual Modeling

One key to the visual modeling approach is to make the programming assignments sufficiently complex, unlike the traditional approach of assigning extremely simple problems to solve, like averaging three numbers. Make the problem trip up even those who have already written programs.

In this way, students more readily see how using a formal process with appropriate problem-solving tools allow them "to work smarter not harder", and increase the likelihood of "doing it right the first time."

Moreover, by asking students to develop software that involves in-depth thinking and providing them with a process (using visual modeling) that adequately support this thinking, the authors have noticed that the advantage of those with prior programming experience disappears! The playing field is leveled for all students.

Another key is to identify an appropriate subset of UML. The authors have found the use-case diagram, flow of events, and activity diagram to be especially useful, taking the student smoothly from problem statement to code.

For example, in designing a "homework help" web page (i.e. provides conversions, like binary to decimal, feet to meters, etc.), the students represent these forms of "help" as use-cases as in Fig. 1. Only then do they propose various solutions. Students see that the use-case is an abstract version of a future solution that allows students to brainstorm later how the solution will be crafted. Instructors just need to augment with additional support that is readily available.

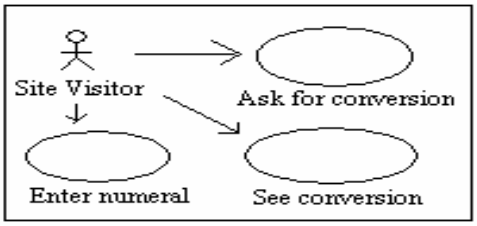

Fig. 1 Example use case diagram

B. Evidence Supporting the Approach

The authors have successfully used the iterative approach with RUP, UML in CS and Software Engineering courses. Based on their positive experiences, described below, the authors feel that formal research by those involved in the computer science education communities is necessary.

Entry Level Computer Science Courses: Over the last four years one of the authors who is also involved in Software Engineering has integrated parts of the RUP and Visual Modeling tools in her courses as a precursor to a formal proof of concept project. She introduces the use-case diagrams initially by asking students to provide a solution to selected problems like the following: "I want to toast bread." The students respond, "Get a toaster." She then says, "OK, I got a toaster, and I see that I must have electricity to use it; but I live in a cave."

Because the students have already invested in part of the solution, they now say that they must wire the cave. Had the students focused initially on the functionality (toast bread) instead of the solution (toaster), they could have proposed not just one but several solutions as they found out more and more about her situation.

With the toaster already in the solution, they had to do the equivalent of "code tinkering" to make this solution work. When students start solving problems using code or even pseudo code instead of focusing on functionality, the structure of the solution is set, just as the toaster set the structure of the solution. That is, students must fix analysis and design problems at the code level.

The activity diagram provided more detail about how the activities would be made to materialize. The students would first write only everyday English into these activity diagrams. Then the students would iteratively re-express the English into paraphrased sentences that finally used only the "verbs" JavaScript or C++.

At that point, the code would write itself, with the tops and bottoms of boxes in the flow diagram becoming the open and closed braces. (Yes, later, much later, the students were told some housekeeping details about omitting certain braces,

indenting, and commenting.) Students then learned visually about creating procedural solutions.

For the part of the course focusing on object oriented solutions and languages, this instructor used part of "Learning to Program with Alice". From their visual storyboards, students created textual storyboards, again starting with given English and then retelling until the code practically wrote itself. The important point is that students focused on solving problems, not coding.

Since the transition from problem statement to code involves no huge gap in thinking between the steps, and the visual models preserve the multidimensional thinking process used, any difficulties that arise are clearly traceable to a particular point in the problem-solving process.

In this way, the students do not fix problems inherent with the design of the program (solution) by tinkering with the code or using blind trial and error. (Instead of buying a toaster for a cave without electricity, they know early that a toaster will not work.) Moreover, students focused on the concepts, no matter the language; in particular they applied concepts before creating the final code:

- The student was able to successfully and efficiently create designs directly translatable to correct code, including nesting and sequencing structures like if and while.
- The Each student was able to explain code and correct his/her own mistakes by referencing the visual models.
- Anyone, including those with coding experience, made moderate to serious errors if they failed to use the process and supporting tools.
- With object oriented programming via Alice, students clearly saw how the storyboards and visual models in Alice related to UML and the overall iterative approach.
- Students gravitated to the visual models and away from pseudocode. Some students in reading the texts would sketch the diagrams in place of the given pseudocode.
- Those with prior programming experience lost their advantage over the others, but all were successful when properly applying the visual models.

Following these courses, former students often returned for help with their first program in the next course. They would have code but no visual models. In no more than 1/2 hour, the student would not only create the visual models but also correct the code themselves. Their failure to rely on visual models in subsequent courses occurred they said because neither the text nor professor used them.

Software Engineering: At the Software Engineering level, once students had developed software using RUP supported by UML, they were amazed with the code they produced. Each semester many of even the top students said that without the visual modeling they would not have produced such high quality code. Given this anecdotal evidence of the effectiveness of a visual problem-solving methodology semester after semester, both at the intro and advanced levels, the authors felt that the computer science profession should look more closely at developing a visual modeling based methodologies to support problem-solving at all levels of software development.

The Professional World: In dealing with co-op and newly graduated students, the authors have much contact with the professional world of software development. Businesses repeatedly state that they need people who can communicate with the outside world and with a gamut of technical people. Students need to be problem-solvers, not just good coders. Students need to innovate and be able to teach themselves. To this end, large companies have their own in house schools. Returning students verify this environment with comments like, "I have been there six months and still have not written a line of code", and "To write any new code instead of using libraries, I have to complete a lengthy form justifying my proposed new code." Many students taking interviews have said, "The fact that I used UML with Rational Rose got me the interview", or "Despite my excellent grades, I never would have been hired but for my RUP and UML experience."

The anecdotal and survey data from the authors' classes and employer contacts indicate that the authors' proposed approach holds promise. Since this approach is amenable to any level of software development, the formal investigation of the effectiveness of this approach in CS education is encouraged.

The next section describes research that indicates why the authors' approach is likely to be validated by this proposed proof of concept research.

III. Need for Proposed Approach

Wide agreement exists that students enrolled in introductory level programming courses should acquire a firm foundation in problem-solving instead of focusing too much on the details of programming languages' syntax. Authors like Coad tried to focus more attention on design [4]. Some, like [19], propose to use Ada to teach problem-solving to non-computer science majors due to the simplicity of the Ada syntax. This simplicity helps students to understand the distinct phase of design method.

Others, like [12], use a spreadsheet/database package as a valuable tool to aid the process of problem-solving. Other approaches [5, 11, 12] involve techniques to improve students' problem-solving by integrating different criteria into an undergraduate computer science introductory course without using any specific programming tools. However, none of these adequately addresses bridging the huge gap between the problem statement and the code. The students still develop their programs at the keyboard and tinker to get the code to work.

That no pervasive problem-solving methodology exists is evident from the contents of textbooks and work presented in papers and conferences. Even the ACM/IEEE CC2001 recommendations and the "algorithms first" approaches do not address the issues related to problem-solving, especially support for modeling. Finally, the method that enables students to perform documentation is missing. One may ask, "Why UML?" The reason is that UML:

- Being a visual-based modeling language, will help to remove ambiguity when analyzing and designing the system,
- Facilitates communication (including with oneself) about the structure during the software development process, from overall functionality down to the code framework,
- Permits use of prior experiences in all walks of life via analogous thinking,
- Preserves visually the thinking process providing students the means to improve their thinking processes.
- Supports modifications in the software structure at every level from describing the functionality to starting coding, and
- Already exists and is an accepted modeling tool. (A few of the UML diagrams, like the class diagram and flow diagram, already appear as isolated pieces in some current texts.)

In summary, UML diagrams serve as a set of progressively more detailed visual models of the software developer's concept of the system to be developed. In this way, UML provides the support for problem-solving and clear communication and also serves as a means for conceptualization using OOA and OOD. In addition, the visual models can be used to offer insights into new OO technologies, like Aspect Oriented Programming [22].

IV. Conclusion

This study reports on using visual tools that enable students to seamlessly progress from the problem statement to the code in beginning and advanced computer science courses. This approach is designed to enhance the quality of students' learning, specifically in the area of problem-solving and programming concepts.

Proper use of UML via hand sketches is adequate, but other aids to visualization are often freely available for educators. Tools like MS Visio [16], Alice, and Rational Rose, will avoid the shortcomings of current approaches to addressing the syntax issues of code. Professors and students will spend more time on problem-solving that results in better coding. While ACM/IEEE 2001 does not mention such an approach to problem-solving at entry level courses, ACM/IEEE does not exclude experimentation with methods. Moreover, we are not proposing changing the ACM/IEEE 2001 body of knowledge.

In addition, the implementation of the proposed methodology does not require massive re-conceptualization of the computer science course offerings, nor does it require that students learn less about core computer science theory while devoting time to visual-based software development skills. We propose using UML to foster, not hinder thinking. Time will be productively spent thinking about the problem instead of trying to fix analysis and design problems at the code level, often by trial and error.

Thus, professors teaching CS will be able to create their own learning environments, using their techniques, but still support and be supported by a unified system of models that facilitate problem-solving and is seamlessly applied throughout the core courses. Moreover, professors in advanced computer science courses will be able to spend more time on topic instead of dealing with recurring coding issues while promoting excellence in computer science.

References

[1] Alice is a 3D Interactive Graphics Programming Environment for Windows 95/98/NT built by the Stage 3 Research Group. Retrieved April March, 20, 2004, from http://www.alice.org/.
[2] Anderson J., & Franceschi, H. (2005). Java 5 Illuminated. Jones and Bartlett.
[3] BlueJ and interactive Java development environment. Retrieved April, 10, 2004 from http://www.bluej.org/,
[4] Coad, P. & Yourdon, E. (1991). Object-Oriented Design. Prentice Hall.
[5] Deek, F.P., McHugh, J.A., Hiltz, S.R., Rotter, N., & Kimmel, H. (1997). On the evaluation of a problem-solving and program development environment. Proceedings of 27th Annual Conference on Frontiers in Education Conference.
[6] Eckel, B. (2003). Thinking in Java, (Third Ed.), Pearson/Prentice-Hall.
[7] Fayad, M.E., Tsai, W.-T., & Fulghum, M.L. (1996). Transition to object-oriented software development. Communication. ACM, 39(2), 108-121.
[8] Felleisen, M., Findler, R.B. , Flatt, M., and Krishnamurthi, S. (2003). How to Design Programs, MIT Press Cambridge.
[9] Guizzardi, G., Pires, L.F., & van Sinderen, M.J. (2002). On the role of domain ontologies in the design of domain-specific visual modeling languages. Invited presentation at Second Workshop on Domain-Specific Visual Languages, 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications. Retrieved April 5, 2005 from http://www.dsmforum.org/events/DSVL02/Guizzardi.pdf).
[10] Holliday, M. & Lugenbuhl, D. (2004). CS1 assessment using memory diagrams. Proceedings of the 26th SIGCSE Technical Symposium on Computer Science Education.
[11] Hyde, D.C., Gay, B.D., and Utter D., (1979). The integration of a problem-solving process in the first course. Proceedings of the 10th SIGCSE Technical Symposium on Computer Science Education.
[12] Kolesar M.V., Allan V.H. (1995). Teaching computer science concepts and problem-solving with a spreadsheet" in Proceedings of the 26th SIGCSE Technical Symposium on Computer Science Education.
[13] Lloyd, B.H., & Gressard, C. (1984). Reliability and factorial validity of computer attitude scales, Educational and Psychological Measurement, 42(2), 501-505.
[14] Naked Objects Framework. (2002). Retrieved April, 12, 2005 from http://www.nakedobjects.org/static.php?content=home.html.
[15] Mahmoud, Q.H., Dobosiewicz, W., & Swayne, D., (2004). Redesigning introductory computer programming with HTML, JavaScript, and Java. in Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education.
[16] Microsoft Visio (2003). Visio Fact Sheet, Retrieved May 1, 2005 http://www.microsoft.com/office/visio/prodinfo/facts.mspx.
[17] Rational Rose. Retrieved April, 20, 2004 from http://www-306.ibm.com/software/rational/sw-atoz/indexR.html.
[18] Savitch, W. (2005). Problem-Solving with C++: The Object of Programming. (Fifth Ed.), Addison-Wesley.
[19] Suchan, W.K. and Smith, T.L. (1997). Using Ada 95 as a tool to teach problem-solving to non-CS majors. in Proceedings of the Conference on TRI-Ada.
[20] Tabrizi, M., Collins, C., Ozan, E., & Li, K. (2004). Implementation of Object-Orientation Using UML in Entry Level Software Development Courses. Proceedings of SIGITE Conference. 128-131.
[21] Ventura, P., & Ramamurthy, B. (2004). Factors that lead to success in CS: Wanted: CS1 students. no experience required. In Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education.
[22] Wikipedia: Aspect-oriented programming (http://en.wikipedia.org/wiki/Aspect-oriented_programming).