

Integrating Visual Modeling throughout the Computer Science Curriculum

Carol B. Collins, 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 memorizing 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, Problem-solving, UML.

I. INTRODUCTION

IN many countries including the 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 [16, 23]. 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.

Manuscript received July 15, 2007, and in its shortened form was submitted on March 8, 2007, as a conference paper (that was accepted).

Carol B. Collins is with the Computer Science Department, 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 the Computer Science Department, East Carolina University, Greenville, NC 27858 USA (e-mail: tabrizim@ecu.edu).

- 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 [5, 10, 11, 17, 22] at computer science and information technology-related conferences.

The symptoms are not an aberration; they are the norm. To convince the students in the software engineering course that the process itself of creating software is indeed linked to software quality, the authors reverse engineered students' cherished "A" rated programs from previous 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 good object-oriented software [8]

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

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 (Association for Computing Machinery) 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. Current Approaches to Addressing the Deficiencies

Three major sources currently offer approaches to address the deficiencies ACM documents, textbooks, and various software aids. Each source seems to fall short of addressing the scope of the problem.

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 generally offer local remedies, but no support for problem-solving in the context of programming, e.g., popular texts like Savitch's [20]. Analysis with real data gets little attention even if textbooks refer to 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 some 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 [7]. 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. The gap from the problem statement to the class diagram is huge. Students who fail to leap this huge gap (and thus leave computer science) may very well succeed with a sequence of smaller gaps. Visual models may be the stepping stones we need to create smaller gaps throughout the curriculum, just as the professional software engineers do.

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 [11] 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 [9]. 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. By using problems that have visual solutions, Alice does seem to contribute to retention and better attitudes as measured on standard scales [13].

C. The Common Weakness of the Current Approaches

As can be seen from the cited specific shortcomings of the current approaches, what is missing in all of these approaches is a unified and formalized methodology that combines a general process for problem-solving with effective tools that:

- support problem-solving irrespective of the code that will ultimately be produced.
- enable students to apply principles and approaches to problem-solving and programming, and
- show the interaction of these approaches, to support, for example, creating modularity of functionality and levels of granularity, and using effectively abstraction, top-down,, divide-and-conquer, foot-in-door.

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 our extensive anecdotal experience over many years, is grounded in a process supported by visual modeling, is broadly applicable, will fit with current textbooks, and will 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 that 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, and include existing partially effective systems (like BlueJ and Alice)

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).

III. 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 [19].

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. For CS1, 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.

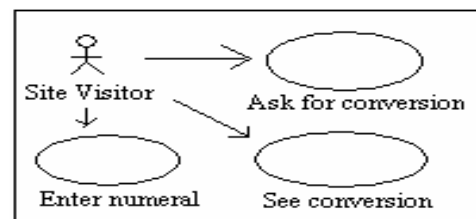


Fig. 1 Example use case diagram

B. Evidence Supporting the Approach

We have successfully used the iterative approach with RUP, UML in Introduction to Computer Science and in Software Engineering courses.

In order to motivate our claim that that formal research by those involved in the computer science education communities is necessary, we discuss here our experiences in three courses, Introduction to Computer Science, Software Engineering I, and Software Engineering II.

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 Introduction to Computer Science courses as a precursor to a formal proof of concept research project.

C. Entry Level Computer Science

In the introductory course, the major focus of RUP is the iterative life-cycle model for which there is a clear continuous trail from inception to final project. Consistent with the level of complexity of the introductory projects assigned, the author selected a few visual models to support student thinking throughout the process: the use case diagram, activity diagram based on flow of events, and the class diagram.

Because newer textbooks discuss the class diagram, we omit describing in detail how these were introduced to the student. However, introductory textbooks do not deal with teaching introductory computer science courses using the other two diagrams. Thus, the authors now describe in some detail about using the use case and activity diagrams and thereby highlight how little additional teaching is needed and how effectively the students use these visual models to avoid

so many difficulties typical to beginning students in computer science.

(1) The Use Case Diagram: The need for requirements analysis involving abstraction is illustrated by asking students to solve a visiting Martian's problem of toasting bread. The students always suggest, "Buy a toaster." The Martian then says, "OK, I got a toaster, and I see that I must plug it into something called a socket; but I live in a cave and cannot find a socket."

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 needed (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 the situation.

With the toaster already in the solution, students 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 use case diagram is presented as a tool to facilitate requirements analysis so that later multiple design features can be considered, like how the functionality is to be modularized to promote ease of maintenance and reuse.

The first assignments are team based. Each team must produce a use case diagram from a given problem statement.

For example, the problem might be "create a web site that will advise an incoming freshman about which math class to take to satisfy the math graduation requirement." First, each team identifies a list of three categories of user (i.e., a professor, parent, enrolled student). Second, each team makes a list of jobs (functionalities) the proposed software will perform, i.e., "The student should be able to view a list of eligible math classes". Third, the team creates a use case diagram reflecting these functionalities. Finally, each team presents the diagram to the class for discussion.

Note that students are provided little instruction. The only instruction given to students for making the diagram from the problem statement is the following applied to one worked example:

- The diagram is to show the jobs of the proposed software but NOT how the system will do the jobs.
- Stick figures (actors) represent entities (human or machine or external database) that interact with the software system to be built.
- The ovals (use cases) represent what the system will do.
- Stick figures are named by a noun that denotes a role, like student, professor, or system administrator.
- Ovals are named by a verb that denotes what the system will do for an actor.
- Lines drawn between an actor and use case represent the actor-use case interaction. If one in this pair always initiates the interaction, the line is drawn as an arrow

with the tip pointing away from the initiator. If either can initiate, no arrow tip is drawn.

Despite this extremely brief set of initial instructions, among the benefits of this visual approach are that student can readily:

- see abstraction at work, allowing an overview of requirements so that big features (like the main requirements and how the functionality is modularized) are clearly visible,
- grasp what the individuals and teams are thinking (i.e., how the problem statement is being interpreted) and thus offer insightful questions and comments,
- see how many different approaches there are to solving a particular problem,
- see how the diagram supports modularization of functionality,
- see how to re-modularize (combine or break up use cases) to improve the solution for usability and maintainability,
- learn, through the discussions, that there often is no single "correct" model and that the "goodness" of a solution depends on many factors.

The most important benefit is that the student is weaned from the professor and starts on the road to self-learning:

- Students can themselves create the diagrams.
- The visualization of the requirements from the problem statement permits any student to understand most of any diagram and thus intelligently discuss it.
- Because all students can discuss any diagram, students themselves correct errors (i.e., misnaming an actor or use case), clarify the requirements, clarify the use case, and understand how to settle on the "best" model to start with.
- As in learning to walk, self correction and improvement is natural: the student sees not only a need for correction but also the approximate magnitude and direction of the correction needed.

Previous coding experience does not help a student in these activities. Even students with no coding experiences can grasp the basics of how the modularity of the use cases affect usability and maintainability in a dynamic situation where requirements constantly change.

(2) The Activity Diagram: The activity diagram provides detail about how the use cases can be made to materialize. Students first provide a narrative of the flow of events, describing the various scenarios by which each use case can unfold from start to finish. From these they make an activity narrative, focusing on the action (verbs) needed to make the events occur.

So that the reader can appreciate how even in CS1 much natural iterative and collaborative problem solving can be conveniently supported by re-expressing these narratives as activity diagrams (instead of pseudo-code, for example) considerable detail is given below about the process the beginning CS students are asked to follow to construct the activity diagram.

For simplicity, only the BOX, IF, IF-ELSE, and WHILE primitive flow structure are used in CS 1 for the activity diagram. The primitives are enclosed in ovals at the CS 1 level to emphasize to the student that the entire structure inside the oval is the primitive. See Fig. 2.

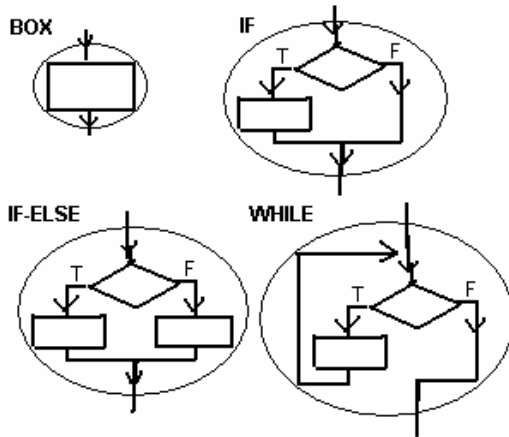


Fig. 2 Four basic primitive flows

For each use case, the student constructs an activity diagram by referring the flow of events and activity narratives (expressed in everyday language). The student (1) chooses and sketches candidate primitives based on the sense of the narratives, (2) copies sentences from the narratives into the boxes and diamonds as appropriate, and (3) iteratively re-expresses these sentences and adding primitives until the activity diagram contains only language that directly translates into a language of choice, like C++ or JavaScript.

This process is smooth and seamless. At each step, the student does exactly what the activity diagram says and verifies that the requirements of the narratives are met up to that point.

Until the final iterations that focus on the chosen computer language, the student needs no coding experience. Thus the focus is on design. Design problems can surface early as students compare early design with the use cases. Often students make changes in the use case diagram, as they see that they should split one use case into two or combine use cases. Students can think this way because the visual tools support this kind of thinking. Even students with no coding experience participate in making such improvements.

The only instruction the student gets for using these primitives to construct the activity diagram is the following applied to one worked example:

- Any primitive (oval) can go inside any box.
- The primitives can be strung together, like electric extension cords, via their entry and exit lines.
- Choose a primitive based on the context of the activity in the narratives; use “while” to repeat an activity, or use “if” to allow for an optional activity, or use “if-else” for choices between two activities, or use a simple box for unconditional activities.
- The activities (repeated, choices, options, unconditional activities) are copied from the initial

narrative and written as complete sentences in the boxes of a chosen primitive.

- The sentence in any diamond is also from the narrative and states the condition that when true will cause the box on the T (true) branch to be executed.
- The contents of each box and each diamond must be a complete sentence (subject, verb).
- Write sentences into any box or any diamond in any order.
- Re-express the English in successive iterations, using more primitives as needed, until the diagram shows that (1) computer will store data needed to carry out the required activities and evaluate the conditions, (2) all verbs are those expressible in the computer language of choice, (3) in the box for the “while” primitive, a sentence changes a stored data value so that eventually the condition in the diamond evaluates to F (false), unless a continuous loop is desired.

Students first practice this process by describing everyday activities, like dressing for the weather (choosing to pick up an umbrella before leaving the house).

Then, students use sequences of flows and nested flows to describe, for example, choosing between (if-else) “eating a dish of ice cream via successive spoonfuls (while)” and “going shopping (simple box)”, whose first pass activity diagram is illustrated in Fig. 3. The “T” branch is the repeated spooning of ice cream into the mouth, and the “F” branch is the shopping activity.

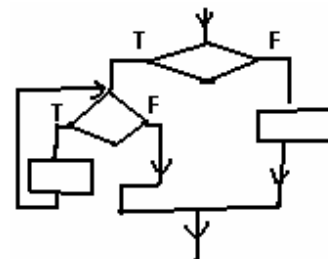


Fig. 3 Nesting primitives

In every case, the emphasis is on choosing the picture and only then filling in the boxes and diamonds. Once the original language is written, and the student checks the diagram against the story, the student may realize that the diagram needs to be modified, for example by adding initial conditions, describing the capacity of the spoon used in eating the ice cream, or describing what the computer must do if the amount left in the dish is less than a spoonful, etc.

Once students are comfortable with activity diagrams for everyday activities, the student gets an assignment that requires a diagram whose sentences are directly translatable to a computer language, like JavaScript.

The student follows the same process, but also paraphrases until the completed primitives are directly translatable into the chosen computer language. Fig. 4 below shows a problem statement with a sequence of activity diagrams (iterations) if the language is to be JavaScript. (Fig. 4 shows the coding step

so that the reader can see how naturally the code is generated by good design.)

Count from 2 to 14 in increments of 3.

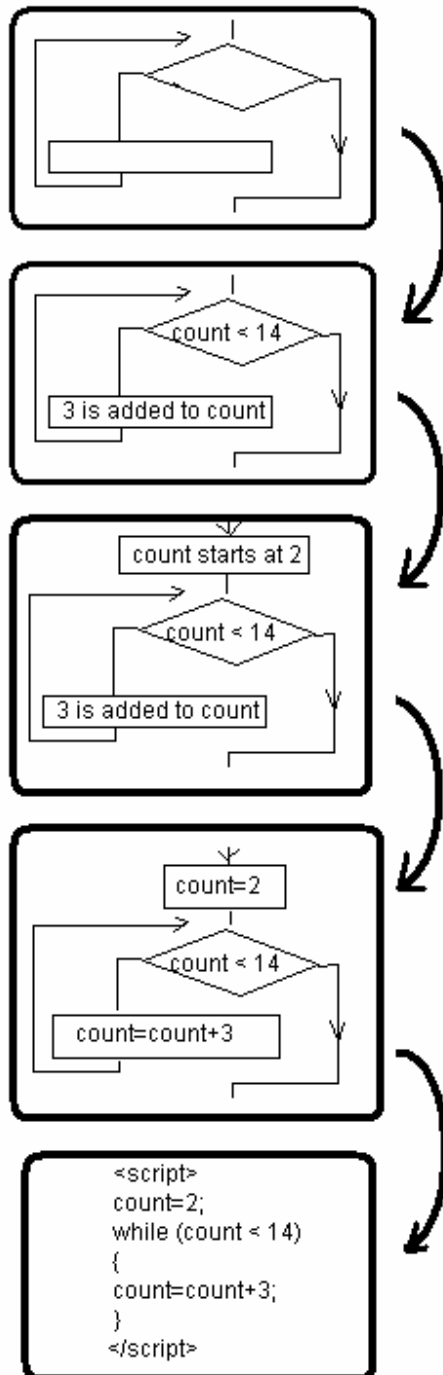


Fig. 4 Example iterations from problem to JavaScript

Only at the last iteration does the student need consider the details of the language, so long as the language considered can express the primitives used in the diagram.

Typically almost all students will produce this particular sequence of designs. Note that students initialize the count only after the third iteration. Their own testing of the design

causes them to add the initialization box; the professor does not tell them to do this. Moreover, if a student initially incorrectly locates the initialization box, the students correct themselves by checking the ‘story’ the activity diagram shows.

Students can do self correcting without the professor, provides the professor refuses to test the design for the student/

Advantages using the activity diagram before coding are numerous for both design and coding.

First, the diagram offers seamless support in drilling down from requirements to code:

- The student never loses his/her train of thought because at any point in time the diagram expresses clearly what the student last thought (interruptions of even days are inconsequential).
- Stepwise refinement with abstraction is supported because the student can check and modify against previous iterations of the diagram that the student already understands.
- Educated guessing and research (as opposed to trial and error) are supported.
- Natural problem solving (a nonlinear activity) is supported, e.g., students acquire and use information as needed, students can clearly see where information is missing, and students have key words to use in research (including asking questions of others).
- The diagram allows students to conveniently find and correct logic errors before investing in code, thereby reducing a source of frustration in coding.
- Collaborative design in teams is enhanced because team members can express their thoughts in a way all team members can understand and explain.
- Students have an overview of the interrelation of design choices and choices in the language chosen as they see that design should smoothly translate to code.

Second, because of the high quality of the design, from which students directly generate code:

- Students generate higher quality code in less time because the features and words in the picture are directly translatable to code.
- Students can explain their code since at the last iteration the diagram is directly translatable into code.
- The activity diagram allows students easily and accurately to incorporate design changes that support code maintenance and may be required by changes in requirements.
- Students correctly locate comments in code.
- Students can differentiate between coding errors and logic errors and thus experience much less frustration in debugging.
- Code tinkering to fix design problems disappears.
- Students learn to reverse engineer, and thus learn new ways of thinking by studying others’ code as they write exiting code into activity diagrams.

- Students are less inclined to mimic code they do not understand.
- Students with coding experience have little or no advantage over others.

Third, and most importantly, the professor is nearly removed from the problem-solving loop, except for the initial set of instructions on the purpose of the diagram and how to make one using primitives.

For example, if a student asks, “Is this design correct?” the professor need only to reply, “Check that the story you told in your diagram expresses the requirements.” If a student asks, “Is my code correct?” the professor need only to say, put the code onto an activity diagram and check this diagram against the requirements.

That is, students learn to be self sufficient learners because they have a tool (effective visual modeling) that supports self sufficiency.

(3.) The Class Diagram: 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.

In addition to activity diagrams, analysis tables (Fig. 5) and class diagrams also help students construct their programs and also understand current programs (more reverse engineering).

Analysis tables organized the objects and their respective methods and properties for the class diagrams. With a list of objects needed to enact activities, the class diagrams shown in the more recent texts in OO programming and the memory maps can make sense to the novice.

Object	Method	Property
frog	hops change(color)	color(green, red)
tree	sways shivers	

Fig. 5 Analysis diagram

Textbooks adequately cover class diagrams, and thus we omit discussing how to introduce these to students. However, the seamless transition from the problem statement to code needs to be maintained. For example, in subsequent courses as complexity of problems increase, other visual models must be added, e.g., the sequence diagram.

In summary, by using an appropriate set of visual models, the student sees that

- the transition from problem statement to code involves no huge gaps in thinking between the steps and iterations,
- the visual models preserve the multidimensional thinking process used, and

- any difficulties that arise are clearly traceable in the visual models 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 focus 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 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 pseudo-code. Some students in reading the texts would sketch the diagrams in place of the given pseudo-code.
- Those with prior programming experience lost their advantage over the others, but all were successful when properly applying the visual models.

Students learned to teach themselves as visual models fostered self-corrective action.

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. Each student said that their own failure to rely on visual models in subsequent courses occurred because neither the text nor professor used them.

However, there is no reason for abandoning visual modeling after the introductory computer science course. By using UML in subsequent courses, the student reviews the visual modeling concepts already learned and merely adds more models to handle additional complexity. With concepts unrelated to code presented independently of code, students have the means to apply these concepts in a wider variety of situations, so that the professor need not re-teach the concepts for each more advanced course involving programming.

D. Software Engineering

The details of introducing software engineering as an advanced undergraduate course are well known. Thus, we omit the details of pedagogy and present only highlights of students’ experiences at our university.

At the Software Engineering level, once students had developed software using RUP supported by UML, they were amazed with the code they produced. In this course 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.

As already mentioned, towards the end of the first software engineering course when students saw their prized projects from other classes reversed engineered into UML class diagrams, the students were appalled at their structures. A typical class diagram consisted of one huge class and two other tiny ones. Moreover, until students saw these visual models, they were unaware of the overall structure of their software from other classes. They readily saw that viewing code was no place to neither understand design issues nor understand how the code related to the original problem statement.

E. *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 and emphasize visual modeling.

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 with the appropriate choice of visual models and the use of visual modeling in common in the computer science profession, 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

IV. 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 [21], 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 [13], use a spreadsheet/database package as a valuable tool to aid the process of problem-solving. Other approaches [6, 12, 13] 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:

- as 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 [24].

V. UNIFYING THE COMPUTER SCIENCE CURRICULUM

Offering appropriate subsets of UML as a tool to support problem solving in programming courses from entry to senior level has clearly been shown to be a possible via the description in this paper of the success of using UML at both the beginning level and senior level of a computer science curriculum.

Professors need not change either the goals or contents of any course, but merely choose the appropriate visualization tools to support problem solving. At the moment, UML

seems to comprise the appropriate set at the moment to support problem solving in the software development context.

If students repeatedly see the same visual tools used throughout the curriculum with more and more tools added as the complexity of the problems increase, students should be able to learn more effectively and to do so as self learners. In this way, students will be better equipped to solve new problems and be more productive professionals, whether in research or in commerce.

That visual models are so useful in learning and problem-solving is not surprising given the research on visualization, like the research presented on the importance of VTN (visual thinking network) in learning and problem-solving, especially in science. See [15] for a discussion of VTN and other issues involving the visualization in problem solving and learning. The research was done with a focus on students trying to learn and apply concepts in science. With VTN Longo, et. Al. [15] assert that "Overtime the novice learner should then have the capacity to transfer this problem solving skill to new situations." Isn't "applying concepts to new situations" at the core of the computer science profession? As they conclude, "The new understanding of the role of the visual cortex in cognitive processing has strong implications for broadening the science education research agenda with respect to research questions, methodology, and foci."

VI. 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 [18], 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.

Finally, appropriate visual models clearly do give students the means to teach themselves and engage more effectively in collaborative learning. Not only do visual models allow students to more easily find errors, but the magnitude and direction of needed corrections become more evident. Communication with others (student team members and professors) about their thought processes is facilitated. If this self sufficiency is begun in CS1 and continued throughout the curriculum, it seems that students may have a greater opportunity for success early and also for greater achievement throughout their academic and professional careers.

The need now is for broadly applied designed experiments, including longitudinal studies, to study systematically the effects of visual based problem-solving begun early and continued throughout the CS curriculum.

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] Collins, C., & Tabrizi, M.H.N, (2007) Using Visual technologies to promote excellence in computer science education. Proceedings of the XXI. International Conference on Computer, Electrical, and Systems Science, and Engineering (CESSE 2007), 21, 83-87, <http://www.waset.org/proceedings/v21/v21-15.pdf>.
- [6] 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.
- [7] Eckel, B. (2003). Thinking in Java, (Third Ed.), Pearson/Prentice-Hall.
- [8] Fayad, M.E., Tsai, W.-T., & Fulghum, M.L. (1996). Transition to object-oriented software development. Communication. ACM, 39(2), 108-121.
- [9] Felleisen, M., FINDER, R.B., Flatt, M., and Krishnamurthi, S. (2003). How to Design Programs, MIT Press Cambridge.
- [10] 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>.
- [11] Holliday, M. & Lugenbuhl, D. (2004). CS1 assessment using memory diagrams. Proceedings of the 26th SIGCSE Technical Symposium on Computer Science Education.
- [12] 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.
- [13] 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.
- [14] Lloyd, B.H., & Gressard, C. (1984). Reliability and factorial validity of computer attitude scales, Educational and Psychological Measurement, 42(2), 501-505.
- [15] Longo, P.J., Anderson, O. R., & Wicht, P. (2002), Visual thinking networking promotes problem solving achievement for 9th grade earth science students, Electronic Journal of Science Education, 7(1), 1-50. http://www.umassd.edu/cas/biology/longo/problem_solving.pdf
- [16] Naked Objects Framework. (2002). Retrieved April, 12, 2005 from <http://www.nakedobjects.org/static.php?content=home.html>.

- [17] 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.
- [18] Microsoft Visio (2003). Visio Fact Sheet, Retrieved May 1, 2005 <http://www.microsoft.com/office/visio/prodinfo/facts.msp>.
- [19] Rational Rose. Retrieved April, 20, 2004 from <http://www-306.ibm.com/software/rational/sw-atoz/indexR.html>.
- [20] Savitch, W. (2005). Problem-Solving with C++: The Object of Programming. (Fifth Ed.), Addison-Wesley.
- [21] 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.
- [22] 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.
- [23] 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.
- [24] Wikipedia: Aspect-oriented programming (http://en.wikipedia.org/wiki/Aspect-oriented_programmig).