TEACHING TECHNIQUES FOR ADVANCED COMPUTER PROGRAMMING

Scott D. Lathrop

This paper was completed and submitted in partial fulfillment of the Master Teacher Program, a two year faculty professional development program conducted by the Center for Teaching Excellence, United States Military Academy, West Point, New York, 2010.

Since some of the courses I teach in the department of Electrical Engineering and Computer Science are focused on programming, I thought I would review what is considered state of the art in teaching computer programming at the university level (versus a trade school or technical certification program). What I hoped to discover was novel classroom techniques that improve someone’s ability to conceptualize and develop a medium to large scale application. What I found is that many scholars advocate non-programming hands-on techniques (Begel et. al. 2004; Budd 2006; McConnell 1996; Pollard & Forbes 2003; Pollard & Duvall 2006) such as games and role playing which emphasize computer science concepts. These scholars argue that the student needs to be removed away from the computer in the classroom (assuming they have access to such resources). While I agree that such non-programming activities are useful, I vehemently disagree that one can learn a procedural skill without actually performing that skill in some manner. Requiring the students to perform this activity outside of the classroom without mentoring, which is often the case, reduces the classroom to a conceptual laboratory only. The instructor really does not have to know or teach any level of detail about the actual technical skill, only the theoretical concepts.

While the techniques explained in the following reviews are interesting, useful, and similar to in class exercises that I and other Computer Science instructors at West Point are practicing (Hill et. al. 2003; Lathrop et. al. 2003), they really dodge the issue of how to help others learn how to program (or other Computer Science technical skills) within the context of gaining a higher level concept (e.g. Object-Oriented, Operating System, Artificial Intelligence). The challenge with teaching a typical computer science course that involves form of programming is that the classroom time and educational experience must balance the teaching of theory and principles with procedural skill. One without the other is setting the students up for failure when the expectation of Computer Science graduates will be that they can think critically in the context of the design and construction of complex software systems. You cannot learn the intricacies of programming by simply role playing, playing games, kinesthetic learning activities, storytelling, etc. as suggested by the reviewed literature without actually hands-on, keyboard programming.

Modeling of a computational process is an abstraction and simplification at some level. Some (e.g. Andrianoff, et. al. 2002; Hill et. al. 2003) acknowledge this limitation but do not address how to overcome it. Astrachan (2004) provides a hint in advocating that an instructor program

---

1 Adapted from USMA EECS Computer Science desired outcomes.
in front of their students. Teaching by demonstration, he states, is an important part of teaching a skill. Yet I find this dissatisfying as then students can simply follow along as “code monkeys” typing away without actually understanding what they are typing. Repenning et. al. (2009) offers another clue in that student-student interaction in the classroom (e.g. “how did you get that 3D model to rotate”), similar to what you see in a middle school computer club, is important for informal verbal feedback and non-intrusive learning. Of course this approach must be tempered with immediate instructor feedback to promote positive techniques and discourage poor design and bad programming habits.

Most authors that suggest hands-on, non-programming classroom techniques realize that it has to be augmented with out of class programming assignments where the students analyze, design, implement, and test an application. However, this activity occurs outside of the classroom without the instructor’s observations. The only observation the instructor makes is of the final product, which by then is too late to capture the student in the moment where deep learning occur. The problem is that this learning does occur but it may not be the type of learning that the instructor desires. This is why I often find one-on-one or one-on-two additional instruction with my students very useful as I can watch them as they are developing their application and monitor their thinking. My driving question then is how do I do a better job of capturing them in their moment of thinking how to organize their code or logic within the classroom?

With that said, I agree with the authors that the non-programming ideas they promote are important to encourage active learning and to move away from lecturing where students are only passively involved in the learning process. Several of the authors (Begel et. al. 2004; Hill 2003; McConnell 1996; Pollard & Forbes 2003; Pollard & Duvall 2006; Schweitzer & Brown 2007) claim that by using these techniques one or more learning styles ((sensory/intuitive, visual/verbal, active/reflective, and sequential/global) are addressed. These ideas, when augmented with programming exercises are invaluable.

**Recommendations**

Based on the review, what follows are a few key points made by the authors:

- Some useful hands-on activities to reinforce a Computer Science concept are games, role playing, playing with toys or other props and storytelling.
- Find a balance between instructor presentations and student activities. In a one hour class the suggested time is 15 minutes of lecture followed by 30 minutes of activity concluding with 15 minutes of discussion and review. Discussing the learning objective both before and more importantly after the activity.
- Establish the learning objective that the activity is supporting prior to developing the in class activity. The activity has to be clearly linked to the lesson objectives.
- Make the activity challenging, yet fun.
- Consider the physical, social, and privacy issues as you develop the activity. For example, consider students with physical disabilities, shy students, activities that may lead to inappropriate touching, or using personal information such as social security numbers or birth dates.
o Keep the implementations of games, role playing exercises, etc. simple. Complex exercises take up valuable time while the students attempt to learn the rules.
o Rehearse the activity to ensure the mechanics of the game do not overwhelm the learning objective.
o Incorporating these activities can be risky as the instructor does not always know which way the students will take the activity. Minimize risk by choosing short, well-structured activities initially, and as your comfort level increases introduce more complex activities.
o Teach programming by demonstration and student-student interaction is an important facet in teaching students to program within the classroom.

Conclusion

Many of the suggestions made by the authors are useful and provide ideas of additional techniques that one can use in the classroom to promote learning—especially of the theoretical and principles of Computer Science. What I have found, however, is that although games, role playing, and other activities prescribed by the authors are important before attempting to delve into the details, more learning occurs after the student has attempted to implement the particular concept and struggled. That is when the analogies start to make more sense and begin to highlight the details of the lesson objectives. This then requires a cyclic form of instruction and a set of lessons where learning objectives are introduced, conceptualized using one or more of the activities discussed in the following reviews, implemented both in and outside of class using the technical tools of the course, and then re-conceptualize again use the techniques advocated by these authors. It also requires some flexibility in the schedule to account for concepts that need to be reiterated. The programming implementation in the classroom at a minimum should be a combination of instructor demonstration along with students working on their own encouraging and discussing with each other when a problem is solved (i.e. “how did you do that?”) with the instructor confirming or denying the student’s approach to limit the practice of bad programming habits. The challenge then for this type of iterative learning is time management both within the classroom and across the semester. Details of teaching advance programming in the classroom are undocumented and should be considered as a subject for future research.

References


**Annotated readings:**


In this paper the authors draw on three separate role playing exercises they developed to introduce OO concepts. They argue that the human is a good metaphor for teaching object systems because like software objects, humans are autonomous, encapsulated actors that interact with other objects. They authors created three different scripts or scenarios using student role playing for introducing OO concepts.

**Script 1:** A role playing activity that emphasizes encapsulation and information hiding. An example script follows: You are an Acrobat. When asked to clap you will be given a number and you are to clap your hands that many times. When asked to do a knee bend you will be given a number and you will stand up and sit down that many times. When you are asked to count, reply with the total number of exercises that you have done. For example, if you have clapped four times and completed two knee bends, you will return six. The instructor then “instantiates” 4-5 “objects” (in this case students) and then begins randomly asking the students to perform the different exercises. This role playing emphasizes that there are multiple instances of objects (students) and that to communicate with these objects you have
to call their “methods” (clap, kneebend, count). These methods may take parameters (e.g. number of claps, number of kneebends) and may return values (number of exercises). This type of scenario is one that the instructor introduces early in a course on object oriented programming.

**Script 2:** The second script comes from a Marine Biology case study that is used in Advance Placement Computer Science curriculum. It demonstrates intercommunication of objects within a moderately complex system. An example scenario follows: You are a Fish and will be given a unique numerical **ID** when the scenario starts. When asked for your **ID**, respond by stating the number aloud. When asked for your **Location**, respond by stating your location aloud. The setup of the scenario continues as described in **Script 1** but with much more complex interactions (e.g. **move** given an **Environment**, **Construct** a **Neighborhood**) between objects. This script in addition to emphasizing the concepts from script 1 also provides a much greater level of inter-object communication, emphasizes the difference between classes and objects, the ability of an object (e.g. an object of class Environment) to pass itself as a parameter to a method (e.g. move), and information hiding.

**Script 3:** The last simulation developed by the authors is a turn-based game featuring two players and a pile of chips. On their turn a player removes chips from the pile. The first player may remove any number, but not all of the chips. Each subsequent player must remove between 1 and twice the number of chips removed by the preceding player. Whoever removes the last chip wins. The software design that the authors use to role play the game includes superclasses and subclasses so that inheritance and polymorphism can be demonstrated. Again, as in the previous scripts, students role play different objects based on the system design. The exercise emphasizes the fact that through inheritance a subclass has all of the capabilities of its parent class—even if it chooses not to use them. It also reinforces concepts such as polymorphism and the model-view-controller paradigm.

The authors encourage discussion both before and after role playing these different scripts as it helps reinforce the concepts being applied. The authors admit that any human modeling of a computational process is an abstraction and simplification at some level and that this debriefing process should be used to highlight these differences. The argue, however, that role playing is beneficial to illustrate flow control in object oriented systems, which is somewhat complex to initially grasp even by more experienced students.


This author reviews the ways Computer Science instructors say they teach programming and provides concrete examples of well-established practices. The author argues that part of teaching programming is to program in front of your students, showing students how you approach a problem. The author states that often times, instructors simply give students a problem or show them some code and say “go”. Just showing students a few snippets of code, he argues, is like a cabinet maker who shows his apprentices a hammer, plane, and a few other tools and then says build a cabinet! Teaching via demonstration is an important part of teaching a skill, which, for computer science, is as important as teaching the theory.

The authors provide examples of kinesthetic learning activities (KLA) defined as “any activity which physically engages students in the learning process” and propose that others in the CS community engage in the discussion. They give examples of two KLA they have used. The first activity is to help students understand the programming language, Scheme’s, primary data structure (a list) and the idea of a *cons* cell. A cons cell has two pointers: a *car* (or *first*) and a *cdr* (or *rest*). For example, assuming a list, l (x, y, z), car(l) returns (x) (a list with one element, x) and cdr(l) returns (y, z)—the rest of the list. A group of five students comes to the front of the room and is given a diagrammatic representation of the list structure on a piece of paper. The group has to physically act out the diagram. Each student represents a cons cell with the left hand being the car pointer and the right hand the cdr pointer. A null pointer is represented with a folded arm. Self-references are pointing at oneself. If a car or cdr pointer points to a value, the student holds a piece of paper with the value written on it. The audience then works out the expression that creates the presented structure.

The authors call the second activity Little People Recursion and centers around trying to learn how to implement recursion (a common programming construct). The instructor poses the problem of calculating factorials and chooses a student to stand and represent the “base case.” That student “knows” that 1! = 1, but nothing more. The instructor selects a second student who’s function is to calculate 2! However, the student only knows that 2! = 2 * 1! so has to ask the first student what 1! is. The instructor then picks students to calculate 3!, 4!, etc. and at each step the instructor has the new student join the others. Again, each student only knows how to calculate n * (n−1)! so in order to solve a factorial problem, they must communicate (simulating a function call) with the student next to them. The point of this exercise is that the nth student only needs the results from the (n−1)st student.

The authors present several advantages and disadvantages to using KLA in the classroom. Advantages include:

1. Engages students by putting them in motion.
2. Reinforces “sensorimotor learning,” in which physical activity transforms into representative mental symbols.
3. Engages other important learning styles, such as Felder and Silverman’s active, sensing, intuitive, visual, or global learners.

Disadvantages include:

1. Social inappropriateness (e.g. touching in the car/cdr Scheme programming language example, sorting students by height or hair length).
2. Privacy issues (e.g. sorting by student ID or social security number or birth date).
3. Physically challenging or have potential to exclude students with physical disabilities.
4. Have potential to exclude shy students or turn them off to learning if not handled properly.
5. Sometimes difficult to manage activities and instructor has to direct and intervene.

The author describes the use of active learning techniques in a programming course. He states that a problem with many programming courses is the lack of hands on activities away from a computer. Rather than spend the entire class period lecturing in a traditional style, the author proposes 15-30 minutes of lecture followed by a daily worksheet. The worksheet is an “anti-quiz” in the authors’ words as students are allowed to work together, ask questions, use the book, and do whatever is necessary to ensure that everybody has the correct answer by the end of the hour. The instructor walks around answering questions and collects the worksheets at the end of the hour to see what the students are not grasping. A follow on homework or project explored the concept in more detail.

An issue the author discusses, was that he previously had prior to instituting what he advocates in the paper, was students completing the course without really knowing how to program. The author concludes that he was giving the students too much of a solution (via code) in class without making them work for it. What is interesting is rather than programming more in class (out of class programming projects were the normal), the author focuses on having the students focus on thinking about the logic of the problem with handwritten notes rather than dealing with compiler syntax issues. The insight was that if he could help students discover what they needed to know, the students would be more confident in their abilities when they sat down to program.

Finally, the author claims the following benefits of his approach:

1) Students spend more time programming. The more experience you have with programming, the more quickly you will become a programmer. *What is strange about this assertion is the author claims not to program in class.*

2) Students gain more confidence in their skills.

3) Students internalize the lessons learned as they have to struggle to arrive at the solution (or partial solution).

4) The instructor has immediate feedback if they gather the worksheet solutions and evaluate them (no grade).

5) Since the worksheets are similar to the exams, the students know what to expect.


John Hill and colleagues advocate the incorporation of different learning styles through puzzles and games. Their goal is to address the different learning styles based on Felder’s model (sensory/intuitive, visual/verbal, active/reflective, and sequential/global). Their games are not only fun but competitive. This paper assesses two such games for an Operating Systems course, a *BattleThreads* game and a *ProcessState Transition* game to help students appreciate different approaches to process and thread management. The authors also provide insights into what makes a makes a good in-class game.
BattleThreads is based on the classic game of battleship. The learning objective is for the students to be able to distinguish the difference between Threads (concurrency within the same process using a shared memory) and Processes (concurrency achieve with multiple processes, each with their own address space) and the advantages/disadvantages of each. The students are broken in to two teams of three to six people. Each team sets up their board as in the classical game of battleship. The board in this case is an actual battleship game, a simulated one, or other classroom props such as the whiteboard. The authors also suggest changing the size of grid based on the number of students to raise the probability of a hit. The game proceeds as in the regular game of battleship where on each turn the current team announces the grid they are targeting. The twist from the conventional game is that the players on one team are Threads and the players on the other side are Processes. This means that for each turn, every player on the team can announce what grid they want to target (concurrency). In addition, the players on the Thread team can immediately see the results of their teammates’ shot as threads share the same memory space. On the other hand, the team members on the other team can only see the results of their shots. To see a teammate’s results, they must use inter-process communication, which was modeled in the game as a rule that a player could either take a shot or communicate the effect of the last shot to the rest of the processes.

In the second game, ProcessState Transition, the learning objectives were to understand how to model process management as the transition of processes between execution states, what data structures are maintained by the operating system to manage these processes, and how scheduling of processes occur. The game is also a role playing exercise. The students, working at a couple of desks, are broken into groups of 4-6 members. Each team is given a game board representing a typical seven-state process transition model (new, ready, ready/suspend, running, blocked, blocked/suspend, dead). One of the students is selected to be the operating system (OS), one is selected to be the timekeeper (TK), and the others become one or more processes. Each process is given a simulated set of processing and blocked times along with its memory requirements that are used in the simulated role playing game. When a process moves into a state which requires memory (e.g. Ready, Running), the player that owns that process must place markers on a board that represents available memory to show how much memory it requires the operating system to allocate it. The students quickly discover the complex management issues an operating system has in deciding what process to schedule, suspend, how it must prioritize certain processes (e.g. system processes) over others, and memory requirements.

The authors assessed these techniques by profiling the students’ individual learning styles using the Index of Learning Styles questionnaire. To evaluate the mastery of the two topics covered by the games, two questions appeared on a major exam. The use of the Process State Transition game did not help student performance. The authors suggest that perhaps it would have been better to assign the student’s roles by taking into consideration their learning style (e.g a global learner be given the role of the operating system process so that they can see the entire picture). The BattleThreads game did not appear to make any difference in mastering learning objective as the students’ scores were comparable to previous years.

However, the students’ self-assessment revealed that they favored this approach to mastering concepts over lecture or discussion. However, several students objected to some of the details that were being overlooked in the games such as the presumed negligible cost of a context
switch, state transition, and virtual memory operations. These details are important when it comes to the implementation of these concepts and what is required if students are to gain a deep understanding of the material.


The authors argue for the use of hands-on laboratories and competitive exercises where students studying computer security and cyber defense related concepts physically install, configure, and maintain operating systems and security (e.g. firewalls, intrusion detection, etc.) software. The students attempt to prevent, and if necessary, react to cyber attacks directed against them. This activity occurs both in the classroom to demonstrate a theoretical concept and hone a skill and also as part of out of class assignments culminating in a hands on “live fire” exercise called the Cyber Defense Exercise. Rather than using role playing, conceptual games, and classroom props, the authors suggest that the deep learning occurs when the students physically type commands at the keyboard, monitor their screens, and see the results of their efforts in either a positive or negative way. Rather than only a in-class game between students of the class, the Cyber Defense Exercise recognizes that fierce competition (at least at this institution) promotes deep learning amongst the students. They not only have to be able to discuss the concepts but actually implement them or fail to succeed in the competition and let their teammates down. This motivates the students to learn the intricacies of the course subject matter.


In this paper the author advocates an active learning style that gets students involved in a classroom activity through reading, writing, discussing, solving a problem, or responding to questions that require more than factual answers. The general idea is to get students thinking about the material mainly because students have a decline in concentration after roughly 10-15 minutes in a 50 minute lecture. One technique the author uses is after 15 minutes of lecture to break the students into pairs and work through problems related to the lesson. For example, some problems may ask leading questions relevant to the material such as, “What will happen if I change this input?” “Why are all four conditions necessary for deadlock to occur?” Other problems may involve one student tracing an algorithm while the second student accounts for object or variable state. In an Object Oriented class, the author has used role playing with props where each object is a paper bag and the “private” attributes are props inside the bag. The students traced code. When objects were instantiated, a student role playing that object would get the bag with all the hidden attributes. Students playing other objects could only access the data through the first object’s “public” methods.

The author evaluated these techniques over a three year period. The first year was no active learning activities, the second year was mostly active learning activities in the classroom (35 minutes in a 50 minute class period), and the third year was slightly less active learning. The
The author gave each group the same final exam (his final exams are not distributed) and used the results of that exam as his benchmark. Furthermore, to avoid difference in grading schemes between years, all exams were duplicated, mixed up (to avoid unconscious biasing of the results) and regraded at the end of the three year period. The author also collected the GPAs of each student prior to his class. The average GPAs were very similar with an F ratio of 0.034—year groups 1 and 3 had slightly higher GPA (0.07 and 0.06 respectively) as compared with year group 2. The incoming abilities did not differ significantly. However, year group 2, had much higher exam scores than year group 1 and slightly higher than year group 3. Significant group activities seemed to help (year group 2 vs. year group 3) but to a much lesser extent. The author’s conclusion is that some activity in the classroom increases learning. He appreciates the need to have a balance between instructor presentations and student activities, where lecture focuses on the critical concepts and the activity reinforces that concept. The author acknowledges that introducing active learning activities in the classroom can be risky at first because you do not know where the class is going to take you. He advises that you can minimize risk by choosing short, well-structure activities initially, and as your comfort level increases introduce riskier strategies.


In this paper the authors advocate introducing teaching techniques reminiscent of kindergarten where the students play games, play with toys, and playing and storytelling. They argue that these techniques promote an active learning environment, level the playing field, motivate students beyond grades, and make class time fun. Their previous work (see next annotated reading below) promoted these ideas in a lab setting. What they propose in this paper is to extend these techniques to the classroom. Games provide a competitive environment which seems to motivate a certain population of students. It is often enough motivation for one student to assist another so that their team “wins.” The authors have used this technique in review lessons. By knowing ahead of time what team each student is on (organized to ensure no team has a major advantage), it encourages the teammates to study together to be prepared for the competition. Assignments that are games lead naturally to competition. The authors have used ConnectFour in a introduction to computer science course for learning 2D arrays. At the end of the assignment, the instructor has the students’ create computer players play each other in ConnectFour to see which team generated the most “intelligent” computer player. The authors state that it is no surprise that many of the *Nifty Assignments* mentioned in the Special Interest Group for Computer Science Education (SIGCSE) are games.

Toys are useful in the classroom for making abstract concepts and theories concrete. For example, a Computer Theory problem is to prove that any $2^n$ by $2^n$ checkerboard with one square removed can covered by L-shaped tiles. The authors used this proof exercise for the past two years, and the students never completed a solution on their own during class time until the authors introduced physical props (a checkerboard, a square “reMOVED” marker, and several L-shapes cut from paper). With this setup, every group except one was able to complete the proof without help. Other toys and props the authors (or those they quote) have used include Frisbees to illustrate parameter passing, beaded blocks for linked lists, playdough with
cookie cutters to illustrate memory allocation, and magnetic letters to illustrate various string functions.

Finally, playing and storytelling engage the students by getting them physically involved in the class, which helps construct cognitive models through action. For example, in an object-oriented class students can represent objects which have to work together to solve a problem. The can act as nodes in a graph or network or objects in a 3D scene that is being ray traced. This type of activity provides an opportunity for the entire class to get involved, obtain immediate feedback, and provide high-order analysis of a concept. Stories help students remember concepts and there is evidence that women are particularly interested in storytelling. An example the authors use is that when they discuss variable scope they say that variables are born in a country and are not allowed to immigrate to other countries. While programming in the lab, one of their students made an error in variable scope. The student’s lab partner, upon discovery of the error said, “Oh! Variables born in Methodpotamia must stay in Methodpotamia.”

The authors discuss some of the pitfalls of these types of activities to include introducing stories that may be socially or culturally offensive; creating physical activities that disabled students may not be able to participate in; and creating overly competitive environments that may turn some students completely off to learning. They argue, however, that the immediate result is a fun, classroom atmosphere. They hypothesize that the long-term effects (the results of which are not published in this paper) will include better overall student performance, deeper learning, diversity, increased enrollments in the computer science discipline. Their long-term goal is to combine these techniques to incorporate more teaching and learning styles and then assess their results.


The authors instruct in an introductory computer science course for non-majors. Previous iterations of the course focused exclusively on either programming skills or problem solving skills that survey computer science. The authors state that their previous experience with teaching the course using a programming language forced the students to learn the particular language and its syntax at a rudimentary level rather than learning the problem-solving principles. On the other hand, a survey course was shallow and did not promote deep learning. So they decided to try hands-on labs where the students work together to solve a computer science type problem using physical props rather than programming. The authors desired deeper learning of various problems across computer science sub-disciplines (e.g. algorithms, artificial intelligence, operating systems, etc.) and to discuss the limitations, critical thinking, logic, and ethical and social issue of the various solutions to problems.

The authors’ methodology was to look at a problem from six perspectives: technical (how something works), virtual (how a problem is modeled or grouped according to its characteristics), theoretical (time and space tradeoffs), social (how life has changed with computers), ethical (legal issues), and philosophical (how minds work; what it means to think, learn, or understand). This approach enabled students to bring their perspective, or academic
strength, to the discussion, enabling them to find a way to use a skill that they were good at and simultaneously gain an appreciation of computer science.

The article discussed three example problems using this methodology. The first problem domain was a robot doing laundry. The props included a checkerboard, randomly placed checkers represented dirty laundry, and a Pez dispenser represented the robot. The robot could “perceive” or not there was a wall in front of it (represented by the edges of the checker board) and it had set of discrete actions to include move (forward/backward), turn (left/right), pick up (laundry), and sleep. With each move action the students could simulate moving the robot forward one square. Working in small groups (3-5), the students’ task was to write an algorithm to get the robot to pick up all the “laundry”, return to where it started, and then go to sleep. The lab instructions encouraged the students to break the problem up into sub-problems (walk-to-wall, turn-corner, etc.), which the instructors hoped would prompt the students to solve a complex problem by breaking it up into components and solving each piece.

The second problem domain was the classic Towers of Hanoi problem where the environment includes three “towers” with the one tower having three or more disks on it ordered with the largest radius disk on the bottom and each disk above it having a smaller radius then the previous disk. The other two towers start empty. The goal is to move all the disks from the starting tower to the goal tower while never placing a larger disk on top of a smaller disk. This is a typical task used to teach recursion and to compare an iterative solution with a recursive solution. The props for this task included a baby’s “ring toy” with three baby “towers”. Again the students worked in small groups (3-5) to try to solve the problem with a hand written algorithm. The first part of lab is to write algorithm for moving 3 or 4 disks, then to move n disks. The “aha” moment is to discover that moving n disks involves moving n-1 disks to the "temporary" tower, and then move the n'th disk to the goal tower and repeat where n = n -1, which involves two recursive calls (something the students have to discover). Questions help guide the students to discover the base case and the students use the props to assist with their analysis.

The third problem domain the authors present is a classic problem typically taught in an operating systems course called the Dining Philosophers Problem where five philosophers sit at a round table doing one of two things: eating or thinking. While eating they are not thinking, and while thinking, they are not eating. A large bowl of rice is in the center of the table. A chopstick is placed in between each pair of adjacent philosophers so each philosopher has one chopstick to his left and right. In order to eat the rice, a philosopher must have two chopsticks. Each philosopher can only use the chopsticks on his immediate left and immediate right. The problem normally illustrates synchronization concepts such as race conditions, deadlock, and starvation.

The authors started this lab by breaking the students into groups of five and having them sit around tables just as in the problem describe above. There are legos in the middle of the table representing rice and utensils (chopsticks) in between each student. Instead of eating, the students must acquire both utensils and then use them to pick up the items in the middle of the table. The first exercise starts with the students grabbing the utensils as quickly or as slowly as they want, but they can only grab one utensil at a time. Then the students are given a deadlock
strategy, then one that does not deadlock, but has the potential of starvation. For each strategy the students must explain how it affects system performance (normally poor). Finally, the group comes up with a strategy that results in neither deadlock nor starvation and present their solutions, which are normally very structured (e.g. only one process can go at a time) to ensure that there is no deadlock or starvation.

After working through the problems in each domain the instructors discuss the technical, virtual, theoretical, social, ethical, and philosophical issues surrounding the problem. For example, in the robot laundry problem the students are forced to think about practical issues such as the shape of room in the “checkerboard” environment versus a real environment; the size and resolution of the squares versus reality; and the lack of static and dynamic obstacles. Social issues such as how household robots would change lives and how the students would or would not trust it for tasks such as babysitting, housework, driving car. Finally they discussed what advances would have to be made to strengthen their trust in an autonomous robot. By discussing the problem from these different perspectives, the students were able to bring their own academic interests into the discussion and discuss the problem from alternate viewpoints. The other problem domains raised similar issues.

The authors conclude that their non-programming, but hands-on lab approach with props and different perspectives had the advantage of emphasizing concept understanding rather than implementation details and allowed the students to understand and apply concepts from more advanced topics in computer science—something they argue many computer science students who have learned to program over a few semesters cannot adequately do because they have been buried in the implementation details. The tradeoff, of course, is that the students who complete this course do not have an appreciation for the amount of work required for implementation of such topics and they also leave the course without any marketable skill.


Repenning, Basawapatna, and Koh state that most university level Computer Science classes do not promote an environment where students can learn from one another. Rather the instructor dominates the discussion and the students have little to no time to interact. They argue for a “middle school computer club” mentality where students work on projects and interact in the classroom when they discover new knowledge or skills. What you want is a situation where one student learns from another by seeing what they are doing and then implementing it. To encourage such interaction both within and outside of the classroom, the authors suggest the following “Flow of inspiration principles”:

1) Display projects in a public forum.
2) Provide the ability for other students to view and run fellow student’s projects.
3) Provide a simple feedback system (e.g. a rating from 1-5 with optional comments) that allows a student to evaluate another student’s project.
4) Allow students time in class to perform these activities providing both verbal and automated feedback.
5) Provide motivation for students to view, download, play, and give feedback on other student’s projects.


This paper illustrates how visualization activities, or computer animations, can be an effective tool to reinforce concepts such as algorithms. The authors also provide some insight as to the important design characteristics of such tools. These characteristics include a high degree of interactivity where the students can set parameters, input, etc.; stepwise control; support for “undo”; and logging of activity so that it can be replayed. The authors recommend that one needs to avoid trying to put too much detail into a visualization tool, but rather keep it short and focused. They also recommend that the interface be mouse-driven as keyboard typing is difficult in a darkened classroom (required to see effectively some of the visualizations). Finally, when visualizing algorithms, they found it better to use a high level of pseudocode with fewer steps and highlight which step of the algorithm is executing.