

AN INSTRUCTIONAL SCAFFOLDING APPROACH TO TEACHING SOFTWARE DESIGN

Stephen Paul Linder
David Abbott
Michael J. Fromberger
Dartmouth College
Hanover, New Hampshire 03755

ABSTRACT

Students often find introductory computer science courses boring and mechanical, leading many to drop from the major. Educators have suggested that bringing realistic design problems into the introductory courses would increase student retention and better prepare students for the major. However, the design and implementation of a solution to a realistic problem is often nontrivial and can therefore be very stressful to students. By using the pedagogical paradigm of scaffolding, this anxiety can be ameliorated. A prototypical design course is described where students implement solutions to progressively more difficult design problems. The solutions to these initial assignments are then continually refactored and used as the code-base for subsequent assignments and the culminating team project. The social interactions necessary for instructional scaffolding are facilitated by having each final project be unique, but similar enough to allow students to help each other. We describe the framework of assignments used in our course, including the capstone projects in which students develop computer vision-based systems that do things like read dice and poker hands, and sort M&Ms.

INTRODUCTION

Software design can be taught at different levels of abstraction, with the lowest levels usually taught in the introductory courses. Low level concepts include language-dependent idioms and object oriented, functional and procedural design patterns such as the design of functions and classes, encapsulation, and abstract data types (Astrachan, Mitchener et al. 1998; Lewis, Rosson et al. 2004). Larger and more interesting architectural design issues are usually postponed until capstone courses. In typical introductory design courses, “*Students get consumed with trying to master the mechanics of programs*” (Denning 2004a). As a result, many do not see the relevance of computer science and over 35% drop from the major (Denning 2004b). Flight from the major can be reduced if introductory courses engage students in motivated inquiry and experimentation, through the design and implementation of solutions to nontrivial problems (Martin 1992; McConnell, Venable et al. 1995; Braught, Miller et al. 2004).

Because real-world design problems often have no obvious or unique solution, there is risk involved in finding the best solution (Coppit and Haddox-Schatz 2005). An effective designer must learn to engage in constructive dialog to craft these solutions (Colwell 2005). Without supportive social interactions, this risk may become unmanageable for students and can lead to indecision and procrastination when students can no longer see an obvious path to a successful solution. Our approach is to incorporate the pedagogical framework of instructional scaffolding, which builds in part on the techniques of cooperative and constructivist learning and authentic assessment.

Instructional scaffolding (Pea 2004; Wyeth and Venz 2004) refers to temporary support structures that teachers provide to help students reach new understandings they could not reach on their own. These supports may be social or cognitive: Properly managed group assignments are one example of social scaffolding, while tiered assignments that build up students' understanding incrementally are an example of cognitive scaffolding. Instructional scaffolding is an application of Vygotsky's *zone of proximal development* (ZPD), which describes the region between the student's individual problem solving ability and the level that individual can reach in collaboration with more capable peers. Vygotsky's social constructivism suggests that learning can be facilitated by a temporary support structure that is retained until the student can achieve results independently (Vygotsky 1978; Allal and Ducrey 2000). Vygotsky believed that social interaction is necessary for effective learning, and that learning happens best when the new concept to be learned is just beyond the student's current reach (Vygotsky 1978). We have extended these principles to the teaching of software development using the Extreme Programming (XP) paradigm.

Cooperative learning is essential to our course. Students should work in teams throughout the course, but substantial care should be taken to promote intra-group and inter-group cooperation and to minimize competition while still guaranteeing some level of individual accountability (Johnson, Johnson et al. 1991). Students in our course begin with small one week assignments in which they are exposed to important design and architectural patterns and build tools that are used in later assignments. As the students become more facile, they work on larger assignments in pairs. In the last half of the course, teams of three students are each assigned a *difficult* and *unique* project that is demonstrated publicly on the last day of the course. The projects are deliberately larger and more complex than one student can do independently, to encourage collaboration. The projects are unique so that groups face no direct competition, but are based on a common technological theme, such as image processing, robotics, or computer networking, so that each project requires many of the same skills to complete. Students are therefore motivated to collaborate with their classmates, but still have the satisfaction of completing a project no other team will be working on. An example of a suite of projects that share many of the same technologies are: the design of a robotic system to sort *M&M's* by color, shown in Figure 1; a backgammon board analyzer, shown in Figure 2, and a robotic arm that can be guide through a maze, shown in Figure 3.

These projects are an example of an *authentic assessment* task (Wiggins 1990). Authentic assessment allows students to demonstrate learning through application of new abilities to a "real-world" situation. Student performance on these real world tasks is not graded in the traditional sense. Instead, students are assessed on their ability to work cooperatively with other students, discuss their project, engage in active enquiry, and publicly demonstrate their work.

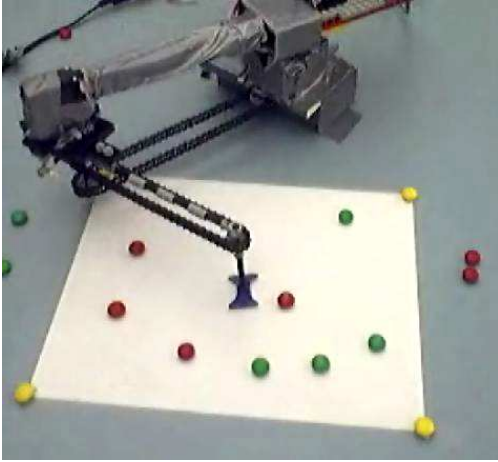


Figure 1. Students were asked to design a robot that would sort *M&M's* by color. Students used a web cam to locate the *M&M's* and the end of a robotic arm. The robotic arm, built with two hobbyist servo motor and Legos, then pushed the green *M&M's* to the left and the red ones to the right.

PROTOTYPICAL COURSE DESCRIPTION

We now present an overview of the introductory design course that has been successfully integrated into the curriculum at both Dartmouth College and SUNY Plattsburgh, and will conclude with an overview of the assignments and projects used for a version of the course based on image processing.

Our course has three major components: classroom activities, short one week assignments, and a final project spanning the last half of the semester.

Classroom Activities

Class time is spent discussing basic engineering design principles. Instead of presenting prepared material about the assignments, the course instructor encourages

the students to initiate all discussion themselves, so that they will begin to ask questions and engage in the social interactions needed for instructional scaffolding and cooperative learning. The volume of material covered is restricted to ensure that students will have enough time for their assignments.

Around the middle of the semester, when students begin working on their final group project, formal class meetings are suspended and the class time is used instead for meetings between the instructor and individual project teams (Adams 1993). Consistent with Extreme Programming principles, teams are required to give demonstrations, and engage in dialog with the instructor about requirements, to ensure that their work is being incrementally developed and continuously integrated. The instructor acts in the role of a *customer*, in supplying the students with the feedback they need to make design decisions; and, at the same time, the instructor acts as a *manager*, in helping to keep the scope of the project reasonable, and to help students avoid dead ends.

Initial Assignments

At the start of the semester students are given short assignments, typically one week in duration, to help them develop software tools, skills, and design paradigms that they can use for their final projects. These assignments have limited scope but are difficult enough that they cannot be completed the night before they are due, so that students learn to leave time to ask questions and explore. These early assignments generally require fewer lines of code per week to be written than the final project, but they require that the students to take substantial amount of time to independently research the necessary algorithms and tools need to complete the assignment. The skills they learn, and the confidence they build completing these initial short assignments help tremendously to alleviate the stress of the final project when they most also work with teammates and give a public demonstration.

In the initial assignments, students modify code that is given to them. They are encouraged to search the web for code or documentation that might help them, so they learn to read code and disassemble programs to find useful pieces and to analyze the design patterns of these programs (Fridley, Jorgensen et al. 1997; Smith 1998). These

assignments are structured so there are multiple solutions, allowing students to experiment with the performance tradeoffs between different designs. As the course progresses, students should be learning to design better and more efficient code by constantly refactoring their code from previous assignments (Adams and Atman 1999). After the initial two or three assignments, students begin working in pairs on assignments, with the pairing changing each week (Srikanth, Williams et al. 2004). This helps the students get to know each other better, and allows them to compare their solutions from previous assignments to find the best starting code-base for the new assignment.

It is imperative for students to use a version control system in their work. Authentic assessment of students is also facilitated with use of a version control system such as CVS¹ or Subversion² and is also consistent with Extreme Programming paradigm:

“Version control is as fundamental to programming as accurate notes about lab procedures are to experimental science. It's what lets you say ‘This is how I produced these results,’ rather than, ‘Um, I think we were using the new algorithm for that graph — I mean, the old new algorithm, not the new new algorithm.’”
(Wilson 2006)

A version control system fosters team ownership and frequent integration of the code. The instructor must convince students of the value of frequent integration by requiring submission of assignments through CVS, requiring multiple versions of the assignments to be tagged, checking that all team members are committing code, and—during longer assignments—checking periodically that the team’s code can be built and run correctly.

Team Projects

The success of a design course depends strongly on the selection of good projects and good teams. The capstone project, like any authentic assessment, should have certain characteristics. It should:

- Be engaging and meaningful, and match the content and outcomes of instruction
- Have real-life applicability
- Allows for multi-staged demonstrations of knowing, knowing why, and knowing how
- Emphasize product and process, conveying that both development and achievement is important (Kerka 1995)

The instructor should avoid having a predetermined design in mind for each project. To foster the kind of interaction and discussion that is most conducive to instructional scaffolding, the instructor should maintain the role of a coach, rather than of an authority. This role is perhaps easiest to play if the instructor has not solved the exact design problem before (Papert 1980; Jonassen 1998).

Team Selection

Research has shown that both team size and team diversity is critical for the success of student projects (Heller and Hollabaugh 1992). We prefer to select diverse

¹ <http://www.nongnu.org/cvs/>

² <http://subversion.tigris.org/>



Figure 2. A camera hung from the ceiling is used to find a backgammon board and locate all the pieces for display on the computer screen. The program gives verbal descriptions of each detected move. Another team was responsible for reading the dice.

a sense of ownership of the project. Projects must be more complex than one student could do alone and have a logical portioning that allows each team member to be responsible for a separate project component. An example of such a project is the automatic recognition of a backgammon game shown in Figure 2. One student can be responsible for locating the board in the image, another the individual pieces, while a third provides a unified user interface with speech output.

Projects should be selected to build on the initial assignments, actively engage the students and create an interesting public demonstration. A public demonstration of the project helps connect the course material to the students' lives outside the classroom—they can invite their friends to see the fruits of their labor, and use their demonstration during a job interview. The projects should interact with the real world and use non-deterministic data as input to promote experimentation and inquiry while requiring students to design and write robust code that is thoroughly tested.

The various projects should be sufficiently different as to foster class creativity and allow each group to “*boldly go where no man has gone before.*” At the same time, they should not be so vastly different that their cohorts cannot understand their project and provide support. Collaboration between groups and *esprit de corps* is fostered because there is no direct competition between groups, and when all the projects are different, teams have no incentive to procrastinate; weaker students and teams cannot afford to wait around for others to lead. Assessment is also made easier with different projects for each team (Daly and Waldron 2004)

Overall evaluation of the project should be based primarily on the quality of the complete work, rather than on the quality or quantity of the underlying code. As with

teams of three students based on information from student surveys, performance in the initial assignments, and the instructor's subjective assessment of personalities. We used the surveys to determine students' schedules, estimate their tendency to procrastinate, and collect their preferences concerning the available projects and team roles. The three team members should have diverse ability levels and should be selected to prevent common problems. For instance, a meek student should not be teamed with two over-bearing students, nor should two procrastinators be put together with a student that likes to start early. The guidelines for group formation in this class closely mirror those that have been used successfully in large enrollment university physics courses.³

Project Selection

Each three person team has a different final project that builds on the tools and techniques learned during the first half of the course. The initial project request is open ended and the form of the final demonstration is continuously negotiated, giving the students

³ <http://groups.physics.umn.edu/physed/Research/CGPS/FAQcps.html>

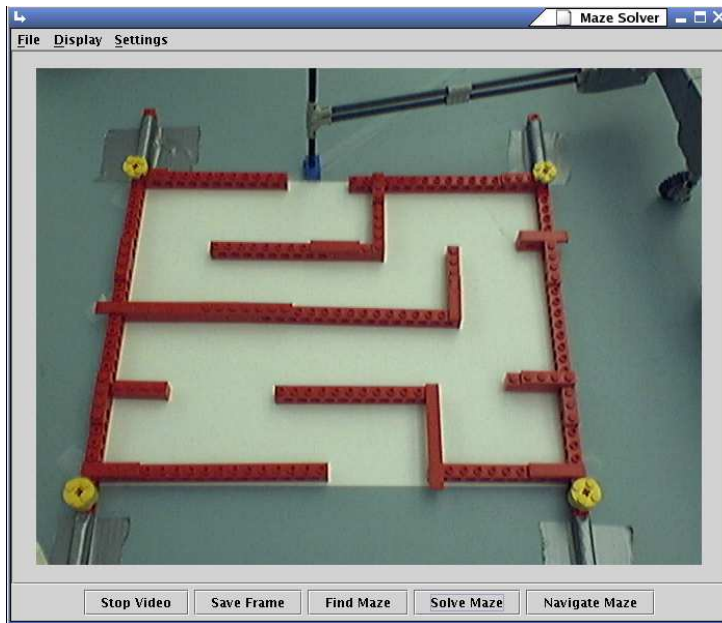


Figure 3. Students were initially asked to push a checker piece through a maze. After much negotiating, the students developed a project that used a visual servo loop to guide the end of a robot arm through the maze, saving all the intermediate way points. They were then able to use the motor angles saved from the way points to then traverse the maze quickly without visual feedback.

any software code base, subsequent refactoring will generally be necessary before the code will become of publishable quality. Therefore the main focus of team meetings should be on what is needed for a robust, coherent and demonstrable project, rather than on minor details of code quality.

EXAMPLE SYLLABUS

The following is a representative syllabus for our course on design principles, based on the version of the course taught during the Spring of 2005 at Dartmouth College.⁴

This is the third course in an introductory sequence. The first course is a standard Java-based programming course. The second course provides the students with a grounding in important basic data structures and the functional programming paradigm, using the Scheme programming language. In addition to teaching software design principles, the course described in this article also provided the students with their first exposure to the Linux environment and the C programming language. Therefore, our initial assignments were chosen to bring students up to speed writing increasingly complex programs in a combination of C and Java, as well as learning how to use important Linux development tools. As additional scaffolding, the students (most of whom had previous background in Java) were allowed to code parts of their solutions in Java, thus taking advantage of their existing knowledge. We also provided them with a number of related example programs they could read and modify.

During 2004 and 2005 the assignments and projects in this course were based on processing and understanding of images captured from inexpensive USB web cameras. This domain allows students to easily interact with the real world and automate simple tasks such as reading dice, counting change, understanding poker hands, observing board games such as Checkers and Backgammon, and controlling computer games with head or hand motion.

Course Meetings and Readings

Readings were assigned before each class meeting. Each class meeting began with a short reading quiz, to provide starting points for discussion and to motivate students to complete the reading before class. The students did not study any formalized process for software development; however, we did discuss its necessity in some domains. Many of the readings were taken from Joel Spolsky's website *Joel on*

⁴ Details can be found at <http://alum.mit.edu/www/spl/Academic/Design>

Software.⁵ Joel writes in an accessible style that demystifies the software industry, writing specifications, software design and marketing, and that raises interesting questions for discussion.

The reading quizzes were supplemented by skill check-off sheets that required the students to demonstrate to a Teaching Assistant (TA) their basic mastery of various development tools, such as the Eclipse IDE, the GNU debugger, *etc.* The check-off sheets forced each student to make time to see a TA and helped open up dialog between the students and the TAs, without the stress of a formal examination. They also served as helpful diagnostics for the instructor as to how the students were progressing in their knowledge of important tools and skills.

Programming Assignments

We gave five assignments designed to help students learn important techniques such as interprocess communication between Java and C programs using sockets, and basic image manipulation tasks. These projects are summarized in Table 1.

The first assignment required students to send one-dimensional arrays of numbers across a socket from a Java program to a C program for sorting. The second assignment builds upon the first by having them send images (two-dimensional arrays of numbers) across a socket to a C program for histogram equalization.

Starting with the third assignment, students were assigned to work in pairs. Students who had trouble with the first two assignments now get a chance to work with another student and see how they manipulated images. This assignment instructs students to read PNG images from files on disk and manipulate them in various ways, which requires them to do research into how to use *libpng*, an open-source library for manipulating PNG images.⁶ The assignment is partitioned into two parts so that each student could be held accountable for an individual contribution.

For the fourth assignment, the students were given new pairings and instructed to read PNG files and detect filled red circles in the images. There are many ways to do this, some more efficient than others. While their initial attempts may not have been efficient, they would later have the opportunity to refactor this code for use in their final team project.

For the fifth assignment, the students were assigned to the teams selected for the final project. This allowed the students to make a more gradual transition into the longer open-ended final project with their team. This assignment required teams to acquire video from a web camera and to find and locate a red ball or similar object in (more or less) real time.

Projects

Students were given a list of team projects to select from and were encouraged at the beginning of the term to submit their own project ideas. The list of projects is changed each semester, so students felt that that they were doing something new, but some themes did recur. Table 2 gives a listing of the five projects completed in the spring of 2005. Three of the five projects used hobbyist servomotors to control small robotic arms. This was done to give students who were also engineering majors a chance to work on a project that used a closed-loop servo system.

Each week, each team scheduled and attended a one-hour “walkthrough” with the course instructor to demonstrate progress and to review documentation and code. The

⁵ <http://www.joelonsoftware.com>

⁶ <http://www.libpng.org/pub/png/libpng.html>

Table 1. A list of assignments given during the Spring 2005 semester. The first two assignments were done individually, the second two in random pairs, and the last one done with the student's team for the final project.

Using Sockets	Send integers over a socket from a Java program to a C-program for sorting, and then back for display on the Java side.
Histogram Equalization	Use a socket to transfer images between a Java and C-program. Convert a Java method that performs histogram equalization to C. Compare results.
Manipulate PNG images	Read, manipulate and save PNG-images using libpng.
Find circles	Find circles in synthetic images and photographs.
Acquire video	Acquire video and find circles in live images

instructor used CVS to monitor the contributions of each team member to the code base. Other impromptu meetings were often held in the lab when the instructor saw students working.

Grading

Each student's final grade was based upon their performance on the reading quizzes, the initial assignments and the team project. On the assignments completed in pairs, both students usually received the same grade, especially if they cooperated well. Because the pairing changed weekly, weaker students could not hide behind a single strong partner for the entire course.

Project grades were based on weekly presentations and discussions with the team, review of the code and documentation archived in CVS, and the public demonstration at the end of the course. The public demonstration was advertised department-wide. Each team made a presentation and demonstrated their working project. The evaluation of the presentation was determined by the scope and difficulty of their project, and whether (and how well) it worked. While students were made aware of the general expectations for the project, and received regular feedback concerning their progress during the term, there was no formal grading rubric for the final project. The grades awarded were uniformly high, given the high level of accomplishment of all students who completed a working project.

Guidelines concerning academic integrity are especially relevant to this course, since students are actively discouraged from writing all the code for their projects from scratch. It is important for the instructor to communicate when code reuse is appropriate (*i.e.*, code from publicly available sources used with proper citation) and inappropriate (*i.e.*, code from current or former students of the college, code written by someone hired by the student, or code used without proper citation). We found that public availability is a good informal metric for whether or not reuse is appropriate, in most cases.

COURSE EVALUATION

The most convincing evidence of the course's success is the public demonstration of the final projects. In the past four semesters during which this course has been offered, every team has demonstrated a working product to peers and non-course faculty members. The list in Table 2 attests to the complexity of the projects undertaken by the enrolled students, most of whom are freshmen and sophomores for whom this is only their third course in computer science.

Table 2. A list of projects done by students during the Spring 2005 semester.

Sorting Marbles	Sorted marbles using a two degree of freedom robot arm.
SpongeBob	Automatically mimic your shuffling of colored sponges using a five degree of freedom robot arm.
Maze Mapping	A robotic arm is guided through a maze using a visual servo loop. Then the arm is quickly returned to the start using the memorized path.
Poker Face	Read a poker hand that is laid on the table in any orientation.
Gesture control of video game	Using hand gestures control a car in a multi-player race simulation.

Although the breadth of content has been reduced as compared to a more traditional software design and implementation course, the instructors of upper division courses have not complained of any adverse effect on the students' programming ability. Indeed, the instructors of the upper division project courses have reported that students are more confident and skillful since the introductory course moved to a project format that included instructional scaffolding, and the students themselves have reported feeling "much more confident" in their programming and problem-solving abilities. The following quotations are from end-of-term evaluation questionnaires given to students in the Spring 2004 offering of the course:

"It has a wonderful new direction and is highly recommended to take. More courses like this one should be offered."

"I learned a lot! Not just specific techniques, but how to independently approach and eventually solve problems I had no clue how to even think about in the beginning."

"[There was] always something challenging to work on, and a sense of accomplishment when finished."

These responses point toward an encouraging level of student enthusiasm.

DISCUSSION

Our design course based on the pedagogy of instructional scaffolding is an effective segue between the highly-constrained assignments often encountered in introductory courses and the more difficult open-ended projects students will see in upper division courses and in the workplace. In our eight years of teaching design, we have found that starting the semester with small individual assignments that build upon each other is much more effective than a single semester long team project. Weaker students build skills early on, and are better positioned to contribute to their team when the projects start. Everyone is held accountable in this approach. Having a different project for each team performs a similar function: Weak individuals and groups can no longer hide behind stronger classmates, while stronger students can veer off on tangents interesting to them without having their results intimidate the rest of the students.

We have also found that the iterative and adaptive techniques of software development in the Extreme Programming paradigm are well suited to an introductory design course. When we first began teaching design in the mid-1990's students followed a waterfall model of software development—they produced long requirement and specification documents at the beginning of the semester and only began coding at

the end of the semester. Usually this approach is only effective if the designer had already solved a similar design problem. Our experience supports this view, and after experimenting with a number of agile development methodologies, we finally settled upon Extreme Programming as the most productive paradigm for the students, as well as the most compatible with our application of instructional scaffolding to the teaching of software design and implementation.

Versions of the course described in the Example Syllabus above have been taught at multiple schools (SUNY Plattsburgh and Dartmouth College), and by multiple instructors, which suggests that its results are transferable between institutions and instructors. It has been our observation, however, that for this approach to teaching software design to be effective requires a strong level of commitment on the part of the instructor, and a “leap of faith” that it will work. The instructor must be aware that the benefits of this approach are diminished if the students are led too strongly toward particular solutions and if the instructor is lured into lecturing, rather than coaching. While more traditional lecture-format courses in software design can be effective, an open-ended cooperative learning framework more effectively promotes learning and the positive benefits of instructional scaffolding and authentic assessment.

REFERENCES

- Adams, E. J. (1993). *A project-intensive software design course*. Proceedings of the twenty-fourth SIGCSE technical symposium on Computer science education, ACM Press, 112 - 116.
- Adams, R. S. and C. J. Atman (1999). *Cognitive processes in iterative design behavior*. Frontiers in Education Conference, 1999. FIE '99. 29th Annual, San Juan.
- Allal, L. and G. P. Ducrey (2000). “Assessment of—or in—the zone of proximal development.” *Learning and Instruction* **10**: 137-152.
- Astrachan, O., G. Mitchener, et al. (1998). *Design patterns: an essential component of CS curricula*. Twenty-ninth SIGCSE technical symposium on Computer science education, Atlanta, Georgia, United States, ACM Press, 153 - 160.
- Brought, G., C. S. Miller, et al. (2004). *Core Empirical Concepts and Skills for Computer Science*. 35th SIGCSE technical symposium on Computer science education, Norfolk, Virginia, USA, ACM Press, 245 - 249.
- Colwell, B. (2005). “Complexity in Design.” *Computer* **38**(10): 10-12.
- Coppit, D. and J. M. Haddox-Schatz (2005). *Large team projects in software engineering courses*. Proceedings of the 36th SIGCSE technical symposium on Computer science education, St. Louis, Missouri, USA, ACM Press.
- Daly, C. and J. Waldron (2004). *Assessing the assessment of programming ability*. Proceedings of the 35th SIGCSE technical symposium on Computer science education, Norfolk, Virginia, USA, ACM Press, 210 - 213.
- Denning, P. (2004a). “The great principles of computing.” *Ubiquity* **4**(48): 1.
- Denning, P. J. (2004b). “The field of programmers myth.” *Communications of the ACM* **47**(7): 15 - 20.
- Fridley, J. L., J. E. Jorgensen, et al. (1997). *Benchmarking: A Process Basis for Teaching Design*. Frontiers in Education Conference, 1997. 27th Annual, Pittsburgh, PA, **2**: 960 - 967.
- Heller, P. M. and M. Hollabaugh (1992). “Teaching problem solving through cooperative grouping. Part 2: Designing problems and structuring group.” *American Journal of Physics Teachers* **60**(7): 637-644.

- Johnson, D. W., R. T. Johnson, et al. (1991). *Cooperative Learning: Increasing College Faculty Instructional Productivity*.
- Jonassen, D. (1998). Design Constructivist learning Environments. *Instructional-Design Theories and Models*. C. M. Reigeluth. Hillsdale, N.J., Lawrence Erlbaum Associates.
- Kerka, S. (1995). Techniques for Authentic Assessment. U. S. D. o. Education, ERIC/ACVE. <http://www.cete.org/acve/docgen.asp?tbl=archive&ID=A032>.
- Lewis, T. L., M. B. Rosson, et al. (2004). *What Do The Experts Say? teaching introductory design from an expert's perspective*. 35th SIGCSE technical symposium on Computer science education, Norfolk, Virginia, USA, ACM Press.
- Martin, F. G. (1992). *Building Robots to Learn Design and Engineering*. Twenty-Second Annual Conference on Frontiers in Education, 1992, 213 - 218.
- McConnell, R. L., W. Venable, et al. (1995). *Freshmen can do rigorous open-ended design*. Frontiers in Education Conference, 1995. Proceedings., 1995, Atlanta, GA, 2: 3c4.1 - 3c4.4.
- Papert, S. (1980). *Mindstorms*. New York, Basic Books.
- Pea, R. D. (2004). "The Social and Technological Dimensions of Scaffolding and Related Theoretical Concepts for Learning, Education, and Human Activity." **13**(3): 423-451.
- Smith, R. P. (1998). "Teaching Design for Assembly Using Product Disassembly." *IEEE Transactions on Education* **41**(1): 50-53.
- Srikanth, H., L. Williams, et al. (2004). *On pair rotation in the computer science course*. 17th Conference on Software Engineering Education and Training, 144 - 149.
- Vygotsky, L. S. (1978). *Mind and society: The development of higher mental processes*. Cambridge, MA, Harvard University Press.
- Wiggins, G. (1990). "The case for authentic assessment." *Practical Assessment, Research & Evaluation* **2**(2): <http://pareonline.net/getvn.asp?v=2&n=2>.
- Wilson, G. (2006). "Where's the Real Bottleneck in Scientific Computing?" *American Scientist* **94**: 5-6.
- Wyeth, P. and M. Venz (2004). *Combining Developmental Theories and Interaction Design Techniques to Inform the Design of Children's Software*. OZCHI 2004 - Supporting Community Interaction: Possibilities and Challenges, Wollongong, Australia.