EXTREME PROGRAMMING FOR SOFTWARE ENGINEERING EDUCATION?

Laurie Williams[1] *and Richard Upchurch*[2]

Abstract *¾ The eXtreme Programming (XP) software development methodology has received considerable attention in recent years. The adherents of XP anecdotally extol its benefits, particularly as a method that is highly responsive to changing customer's desires. While XP has acquired numerous vocal advocates, the interactions and dependencies between XP practices have not been adequately studied. Good software engineering practice requires expertise in a complex set of activities that involve the intellectual skills of planning, designing, evaluating, and revising. We explore the practices of XP in the context of software engineering education. To do so, we must examine the practices of XP as they influence the acquisition of software engineering skills. The practices of XP, in combination or isolation, may provide critical features to aid or hinder the development of increasingly capable practitioners. This paper evaluates the practices of XP in the context of acquiring these necessary Software Engineering skills.*

Index Terms ¾ Extreme Programming. Pair Programming, Software Engineering, XP

INTRODUCTION

Extreme programming [1, 2] (XP), introduced in 1996, is a lightweight, yet disciplined software development methodology. Although it departs significantly from traditional development practices, anecdotally, XP appears to be effective. Industrial interest in the use of the methodology is growing very rapidly. Computer science educators around the country are also expressing interest in applying XP in educational settings. Most of this interest is sparked by anecdotal evidence from industry extolling the benefits of the practice. Some educators, including the first author, have already introduced the methodology in Software Engineering undergraduate courses.

This paper consists of five sections. The first section is an overview of the XP methodology and associated practices is next presented. The second section is an overview of instructional models for educating competent software engineers. Then, the development of intellectual skills is discussed and the XP method as an educational practice in software engineering education is examined. In the last section, the potential XP may have in the education domain is examined.

EXTREME PROGRAMMING PRACTICES

Industrial strength practices are one source for potential activities to support student development. Recently eXtreme Programming [1-4] (XP) has gained the attention of the software development community. Interest from the development community is sparked by anecdotal evidence extolling the benefits of the practice in terms of staff morale, reduced project schedules, and satisfied customers. Table 1 [5] provides an overview of the practices associated with XP. Each of these will be briefly described below.

TABLE I
XP PRACTICES

Metaphor	Unit Testing
Collective Code Ownership	Functional Test
Simple Design	Pair Programming
Refactoring	Coding Standards
Small Releases	Open Workspace
Continuous Integration	40-hour week
On-Site Customer	Planning Game

Metaphor. XP believes that each application should have conceptual integrity based on a simple metaphor, which explains the essence of how the system works. For example, one large XP project was a payroll system for Chrysler. The metaphor for this project was that the payroll system was like an assembly line where hour parts were converted to dollar parts, all parts were assembled and a paycheck was produced [5].

Collective Code Ownership. On an XP development team, no single programmer 'owns' any part of the code. The code is entered into the team's collective code base. Once entered in the code base, every member of the team owns the code. Then, any member of the team is able to change any code in the code base without asking for 'permission' from anyone.

Simple Design. XP strives for supremely simple designs. They stress that programmers should not try to predict future needs and to design accordingly. They have two tenets to support their design philosophy: "You aren't gonna need it." (or YAGNI) and "Do the simplest thing that could possibly work."

[1]Laurie Williams, Department of Computer Science, North Carolina State University, Raleigh, NC 27695, williams@csc.ncsu.edu
[2] Richard Upchurch, Computer and Information Science Department, University of Massachusetts Dartmouth, N. Dartmouth, MA 02747, rupchurch@umassd.edu

Refactoring. Refactoring is the process of improving the code's structure while preserving (not improving) its function [3]. XP advocates refactoring code continuously and explicitly.

Small Releases. XP heightens the pace of spiral development by having short releases of 3-4 weeks. At the end of each release, the customer reviews the interim product, identify defects, and adjust future requirements.

Continuous Integration. Coding assignments are broken up into small tasks, preferably of no more than one day. When each task is completed, it is integrated into the collective code base. As a result, there are many product builds each day.

On-site Customer. The customers are always readily available and accessible to the developers for the purpose of clarifying and validating requirements throughout the implementation process; preferably, customers are on-site.

Unit Testing. Extensive, automated white box test cases are written before production code is produced. These automated tests are added to the code base. Before a programmer can integrate their code into the code base, they must pass 100 % of their own test cases and 100% of every test that was ever written on the code base. This ensures that the new code implements the new functionality without breaking anyone else's code.

Functional Test. Traditionally, project management techniques have been based on a developer's own assessment of how much of their task has been completed. Alternately, XP promotes the use of functional test case tracking for calculating project completeness. XP terms this assessment "Project Velocity." Functional test cases are based on customer scenarios. When a functional test case is successfully passed, it can be considered that a specified functionality has been implemented properly. Project completeness is based on the percentage of functional test cases that have been passed. Team members can unequivocally compute this measure.

Pair Programming. At all times, two programmers work side-by-side at one computer, collaborating on the same design, algorithm, code or test.

Coding Standard. In order for developers to easily understand each other's code, an agreed upon coding standard is followed. Pair programming and collective code ownership makes the use of a coding standard very important.

Open Workspace. Personal communication between developers and customers is paramount in XP. Workplace layouts have common areas that facilitate open communication.

40-Hour Week. XP advocates that programmers do not tire themselves out by overworking themselves. They have found that during crunch periods when overtime is worked, the artifacts that are produced are poor.

Planning Game. The techniques for gathering requirements in XP are a radical departure from that of more traditional software methodologies. First, customer requirements are written in natural language, informal "User Story" cards, similar to use cases [6]. These cards are never formalized, no relationships or dependencies between the cards are identified. Software developers place time estimates and customers assign priorities to each card. Together, the developers and the customers play the "Planning Game" in which the customer chooses those User Stories that comprise the most important content for a short, incremental deliverable of about one month. Each short implementation increment is accepted and tried by the customer. Then, the remaining User Stories are re-examined for possible requirement and/or priority changes and the Planning Game is re-played for the next implementation increment.

INSTRUCTIONAL MODELS FOR SOFTWARE COMPETENCE

Many agree that a primary factor in producing quality software is excellent designers [7-10]. However, The development of competent to excellent software practitioners remains a challenge. Software engineering education (SEE) seeks to identify those critical ingredients that result in competence in the field, then to develop instructional models that prepare students to become effective practitioners.

Cognitive research has revealed that developing intellectual skills, such as those associated with software engineering, requires explicit instruction and carefully constructed practice in the context in which such skills will be applied. We discuss four different strategies to improve the number and quality of skilled designers graduating from our educational programs.

Early Identification

One approach is to identify top designers as early as possible, then nurture these individuals over time [11]. The problem with this strategy is that those skills that indicate early promise are often not the same characteristics that promote mature expertise. Evidence from human resources research in industry suggests that initial ability predicts entry-level performance, but does not predict long-term job success [12]. In addition, grade point average is a very weak predictor of later success [13]. Although most of this research has been on domains other than software development or programming, similar conditions prevail in software engineering. As a result, we do not consider early identification a salient strategy.

Design Knowledge

A second strategy focuses on increasing our instruction of design principles and design artifacts. Here our attention is on identifying the characteristics of good programs such as encapsulation, information hiding, and modularity [14]. Such principles are useful and easily incorporated into instruction. This approach assumes that a critical factor for differentiating excellent designers from others is their knowledge. Unfortunately, knowing appears to be different from applying. Hence, knowledge of these constructs does not automatically enable designers to incorporate them into their own designs. Teaching declarative knowledge alone is not sufficient because it focuses primarily on the product of design. The approach does not provide an opportunity for students to develop how-to knowledge essential to conduct the business of design [15].

Practicum

The practicum is a third approach to helping students develop design skills. In this approach, a realistic environment is provided in which students learn the skills they will use in the "real world" [16]. The activity is usually in the form of a project. This approach acknowledges that the skills needed to be effective software engineers are not limited to declarative knowledge or to programming skills per se. Usually there is little effort in identifying what those other skills may be. Rather the approach assumes that embedding students in an authentic context provides sufficient opportunity to develop these unexplored skills. This approach is based on the situated cognition principle (or learning within an authentic context), which claims that expertise is domain-specific and can only be acquired in the context in which it will be practiced. Typically, however, instructors focus on the experiences with little regard to what is actually learned from those experiences (a criticism leveled at engineering design education in general [17]). Yet, it is predicated on the assumption that experience alone is sufficient to develop expertise.

However, experience alone is a weak predictor of future performance [18]. Practice alone is sufficient to develop a certain level of competence, but, for most people, will not result in expert levels of performance. To develop to higher levels of competence, deliberate practice techniques are necessary [19]. Deliberate practice involves work on particular skills in an effort to improve. Having students engage in a few large-scale projects during their undergraduate experience gives them a limited amount of practice. It does not, however, help them develop the skills necessary for deliberate practice. Furthermore, the expertise literature 20] indicates that novices' understanding is tightly coupled to the context in which the material is learned. Thus, novices are able to apply what they have learned only to very similar situations. In contrast, experts are able to apply what they know to a broader range of problems. Experts evidently develop abstracted (context-independent) representations as a result of repeated experiences in different contexts. Hence, a single project is an insufficient basis on which to develop an adequate representation of the necessary knowledge, and we should not expect much transfer from the project activity to other design activities unless they are very similar.

Process Knowledge

A fourth educational approach to software engineering education emerged by explicitly identifying the processes needed to do design and teaching those skills directly [21-23]. To implement this, educators must first understand the processes underlying the development of effective software, identify the subskills and roles involved, and then construct situations that require students to practice those skills. A process focus in SEE has several important attributes. First, the concern for process forces an early delineation of subprocesses, which in turn helps to isolate the necessary subskills. By decomposing the students' project development into more discrete steps, with higher granularity, the deliberate practice needed to develop the competence needed in our graduates can be achieved. Second, the identification of subskills provides an opportunity for modeling and coaching, which have been shown to be a powerful instructional strategy for cognitively demanding tasks including software design [24-26]. The process focus in SEE seems to combine the necessary ingredients for developing some of the heuristic strategies needed in design practice.

Instructional Model Analysis

Our analysis indicates several things: 1) if we wish to produce skilled designers then our educational programs must attend explicitly to that 2) producing competent designers requires teaching both the knowing-what and knowing-how, and 3) knowing-how knowledge can only be taught in an environment in which the student actively engages in appropriate activities. We believe the process focus has the right characteristics needed for the next generation SEE. It remains to be determined what processes produce good educational outcomes [27]. Yet, clearly the above discussion focuses our attention of identifying how a given practice, or set of practices, provide the learner with those opportunities to develop the requisite skills for ongoing learning and development.

INTELLECTUAL SKILLS DEVELOPMENT

Knowledge can be decomposed into three distinctive categories: (a) declarative knowledge ("knowledge that"), (b) procedural knowledge ("how to knowledge"), and (c) metacognitive knowledge (self-monitoring, agency, reflection). Declarative knowledge refers to the kind of knowledge typically learned from textbooks--facts and concepts. Procedural knowledge refers to being able to do something, such as write. Metacognitive knowledge refers to a person's skill at planning strategy, monitoring process and progress, changing what one is doing when appropriate, and reflecting on the process. Yet, the learning activities in our courses typically consist of reading textbooks, listening to lectures, and taking exams on the material; this kind of learning is declarative. Programming assignments usually augment in-class material; this requires procedural learning. When programming assignments are merely added as homework assignments, the implicit assumption is that declarative knowledge is a sufficient basis for procedural learning to occur.

Research within the software domain, and in other fields, makes it clear that each knowledge category must be addressed explicitly in instruction. For example, studying instructional text is not a sufficient basis for students to solve LISP programs, whereas *doing* one programming problem improves the probability of successfully completing a second one by 50% [28]; adding an instructional example of how to construct the program produce improvements of over 60% [29]. It is equally true that one cannot teach procedural skill by teaching procedural principles. For example, [30] found that teaching students the principles of top-down design was not sufficient to enable them to practice top-down design.

Although less studied, a similar claim can be made for metacognitive knowledge. When solving problems, most university students do not spontaneously consider strategies, plan their approach, evaluate their progress, and think through how to change what they are doing. When asked to think metacognitively at every step of a problem-solving episode, such students can do so, and, through doing so, develop deeper understanding and better performance on subsequent problems [31]. Studies in programming domain have demonstrated that (a) students who reflect on what they are learning learn better both on declarative and procedural tasks [32], and (b) inducing students to reflect upon the material is effective, suggesting that it is the metacognitive activity that produces the improved performance [33].

The principled focus on the metacognitive skills appears to be one of the keys to facilitating the evolution of higher levels of competence [34]. Students learn to think about how and why they work in particular ways, and how to develop strategies for altering ineffective habits. This promotes the development of representations of declarative/procedural schemas that appear to be essential in expertise. Hence, our approach to assessing XP practices must focus on how each practice may facilitate the acquisition of those skills that appear to be critical in the development of enhanced competence.

XP AND DEVELOPING EXPERTISE

From our analysis, as given in the discussion above, there are a number of distinct items which should be present in an educational setting for the acquisition of skills which may lead to exceptional levels of performance. Clearly it is not sufficient to simply adopt industrial strength practices. The practices incorporated into our pedagogy must accentuate those skills and abilities viewed as critical. Our approach to assessing XP practices must focus on how the practices associated with this development method may facilitate the acquisition of the intellectual skills.

As a cautionary note, it has been said "XP is aimed primarily at object-oriented projects using teams of a dozen or fewer programmers in one location [5]." Since these parameters comprise a relatively small percentage of industry projects, we focus on teaching our students the right skills to handle projects with varying project parameters. We believe that several of the XP practices are very valuable in the context of a general SEE. However, some universities require students to take only one software engineering course. If this singular course only used XP, students would lack the skills for documenting and designing larger projects. XP practitioners and researchers are working towards adapting XP towards larger, perhaps geographically distributed project teams. However, the XP presented in this paper is intended to handle the smaller teams as described in the quote above.

The fact that XP uses a defined process that structures the development activities is a step in the right direction. Students need to be provided with a defined process and helped to understand their role in the process. An effective software developer a) understands the development process or processes; b) conceptualizes a desired process; c) establishes process improvement actions; d) plans the improvement activities; e) finds the resources needed by the plan; f) executes the plan; and g) repeats the improvement process [35]. Embedding student development in a defined process provides opportunities for a process focus considered critical for modeling and coaching the subskills needed for software development. Furthermore, the process focus provides access to those skills needed to empower students to become increasingly independent as they move toward higher levels of competence where they manage and evaluate their personal development strategies. Lastly, when students are required to use 'heavyweight' processes that require much documentation, they often develop distaste for software development processes. Most quickly revert to ad hoc procedures when not explicitly required to follow a process. XP may provide a process that students will not reject.

The practices of XP will now be re-examined to provide guidance on the use of the practice in an educational context, considering the use of XP for a student project team.

Metaphor. The idea of a system metaphor is very abstract and often mysterious to experienced practitioners. It is probably best not to stress this aspect of XP, particularly in the undergraduate classroom.

Collective Code Ownership. An XP practice that allows collective code ownership is the extensive automated unit tests and the criteria that 100% of these test cases must pass prior to code being integrated into the code base. Then, when a programmer changes someone else's code, they can be assured they did not break previously implemented functionality. Unless the students are relatively experienced testers, collective code ownership will likely present problems within the group.

Simple Design. XP's emphasis on simple design is powerful. Students sometimes deve lop a 'macho' attitude toward program, whereby the 'smart' programmers take pride in developing code only they understand. Designing and implementing simply, such that their pair-programming partner can continually understand will likely breed better practices with out students. However, XP also does not require any form of written design documents. We believe it is essential that students learn design practices such as use cases [36], UML [37], and CRC cards [38]. Most XP practitioners know these design practices and can decide to use them as the need arises. We need to insure our students do too. Therefore, we advise that students are exposed to these practices in the curriculum.

Refactoring. Refactoring is a wonderful practice to teach students in a Software Engineering course. Students, particularly those who are relatively new to a programming language, can write "smelly code" (a term used by Martin Fowler [4]). It is great practice for the students to re-implement smelly code given by the instructor with better structure, without changing the functionality.

Small Releases. What excites many about XP, and which may have a similar impact on SEE, is small releases [25]. Product development based on small releases implies developers reap the benefits of frequent feedback. In the educational setting this feedback is precisely what is required for helping students make their development process visible through measurement activities. Like the Team Software Process^{SM} (TSP^{SM}) [39], which cycles through development over a short period of time, small releases allow exposure to the full cycle of development repeatedly. This repeated exposure provides the opportunity to again focus students on learning those subskills of planning and evaluating.

Continuous Integration. Students often underestimate the difficulties of integrating code. Having very frequent integrations, along with a defined configuration management strategy, is an excellent lesson for the students.

On-Site Customer. Choosing projects with "real" customers has important benefits in SEE. There are numerous issues dealing with communication skills that arise with this practice. The instructor can serve the role of the customer and responsively clarify project requirements.

Unit Testing. Students can certainly benefit from XP's unit testing procedures. A popular shareware unit testing tool for Java is Junit (available at http://junit.org) while similar tools are available for other programming languages (see http://www.xprogramming.com/testing.html).

Functional Testing. As unit testing provides students with excellent white box testing experience, XP's functional test procedures are beneficial for learning black box testing.

Pair Programming. There is a growing body of empirical evidence related to the efficacy of pair-programming as an educational practice [40]. We have outlined elsewhere [41] how the practice of pair-programming influences the outcomes in the software engineering classroom in terms of satisfaction, problem solving, learning and team building and communication.

Coding Standards. Many software engineering classes already require the use of coding standards. The use of pair programming, as well as other XP practices, provides more incentive for the students to follow the standard.

Open Workplace. Student labs have long provided open, collaborative environments. Trends towards students owning their own computers and working in their own room have created a less collaborative environment for them. Working in an XP team would require the students to find a common workplace, which is beneficial when compared with a student team project where each team member works on their own part in their own room and integrates the pieces late into the process.

40-Hour Workweek. Though it may seem ridiculous to consider the 40-hour week as an educational practice. Yet, our current approaches merely fuel the current practices where heroics, based on overtime and late nights, are often valued over reasoned and deliberate practices. In some regard the academic world has promoted a view of the software developer as the nocturnal loner who thrives on generating code to meet the critical deadlines. This perspective needs a serious revision. Students need to gain more control over their development activities through time management, hence encouraging planning and coupling that planning with a defined process should help lead them to a more disciplined approach to the activity.

Planning Game. Planning and estimating is an important component of XP development. Developers make explicit choices on which parts of the system they will work. As part of this, developers estimate the time required to complete it. Coupled with small releases, the planning and estimating activities provide significant feedback to the novice developer. In the educational context, with the iterative development inherent in small releases we can implement reflective experiences where the student is directed to evaluate the result of the activity in terms of their accuracy in estimating and those characteristics of their process that contributed to their ability to meet their estimate.

CONCLUSIONS

Let us first assert that SEE requires the same solid theoretical and empirical foundations as software engineering. What we have provided are conjectures related to how the practices of XP *may* influence learning in the software engineering program.

Part of the missing ingredients in SEE is a lack of visibility of what is actually learned. We evaluate student products and claim success based on the quality of those products, yet we cannot state with any certainty how those products came into existence. Nor can we claim any true understanding of the state of our students' knowledge of the field. Achieving expertise, or higher level of competence, requires deliberate acts that help students attend to their evolving skills acquisition. As such, methods to evaluate their progress towards this goal need to be developed. In other design disciplines, there are efforts underway to assess the nature of learning and then evaluate the manner in which this learning changes over time. One such approach, structural assessment [42], attempts to evaluate a student's knowledge of the relationships among concepts in a domain.

We briefly discussed TSP and XP. We would like to conduct a serious empirical comparison of these two approaches. This should be conducted using methods such as structural assessment to uncover any critical differences that may result from these approaches.

XP offers potential benefits for SEE; the practices offer much pedagogical appeal. We encourage others to carefully implement these practices and lend their results to the experience base.

REFERENCES

1. Beck, K., *Extreme Programming Explained: Embrace Change*. Reading, Massachusetts: Addison-Wesley, 2000.
2. Beck, K., & Fowler, M., *Planning Extreme Programming*. Reading, Massachusetts: Addison-Wesley, 2001.
3. Jeffries, R., Anderson, A., & Hendrickson, C., *Extreme Programming Installed*, Reading, Massachusetts, Addison-Wesley, 2001.
4. Fowler, M., *Refactoring: Improving the Design of Existing Code*, Reading, Massachusetts: Addison-Wesley, 1999.
5. IEEE Computer Society, XP Practices, DynaBook: eXtreme Programming, http://computer.org/seweb/Dynabook/XPPracSdb.htm.
6. Jacobson,I., *Object-Oriented Software Engineering: A Use Case Driven Approach*, ACM Press, 1992.
7. Brooks, F. P., "No Silver Bullet," *IEEE Computer*, 1987, pp. 10-19.
8. Card, D. N., McGarry, F. E. & Page, G. T., "Evaluating Software Engineering Technologies," *IEEE Trans. on Software Engineering, SE-13*, 1987, pp. 845-851.
9. Yourdon, E., *Rise & Resurrection of the American Programmer*, Englewood Cliffs: Prentice-Hall, 1996.
10. Curtis, B. "What if programmers were treated like jocks?" *American Programmer*, 10(5), May 1997, pp. 9-12.
11. Freeman, P., "Essential Elements of Software Engineering Education," *IEEE Trans. on Software Engineering, SE-13*, 1987, pp. 1143-1148.
12. Schmidt, F. L., & Hunter, J. E., "Development of a causal model of processes determining job performance," *Current Directions in Psychological Science, 1*, 1992, pp. 89-92.
13. Erhmann, S., "*ASKING THE RIGHT QUESTION: What Does* Research Tell Us About Technology and Higher Learning*?*" The Annenberg/CPB
Projects, 1995. Available
http://www.learner.org/edtech/rscheval/rightquestion.html.
14. Buschmann, F., Meunier, R., Rohnert, H, Sommerlad, P. & Stal, M., *Pattern-Oriented Software Architecture: A System of Patterns*.
15. Davies, S. P. "The Nature and Development of Programming Plans," *International Journal of Man-Machine Studies, 32*, 1990, pp. 461-481.
16. Denning, P. J. "Educating the New Engineer," *CACM*, 35, 1992, pp. 83-97.
17. Dixon, J. R. "The State of Education," *Mechanical Engineering*, February 1991, pp. 64-67.
18. Ericsson, K. A., Krampe, R., & Tesch-Röhmer, C. "The Role of Deliberate Practice in the Acquisition of Expert Performance," *Psychological Review, 3*, 1993, pp. 363-406.
19. Ericsson, K. A. (ed.) *The Road to Excellence*. Mahwah, NJ: Lawrence Erlbaum, 1996.
20. Ericsson, K. A. "The Acquisition of Expert Performance: An Introduction to Some of the Issues," K. A. Ericsson (ed.) *The Road to Excellence*. Mahwah, NJ: Lawrence Erlbaum, 1996.
21. Humphrey, W. S. *Introduction to the Personal Software Process*. Reading, MA: Addison Wesley, 1997.
22. Upchurch, R., & Sims-Knight, J. E. "Designing Process-Based Software Curriculum," Proceedings of the Tenth Conference on Software Education and Training, Virginia Beach, VA, April 13-16, 1997.
23. Humphrey, W. S., *A Discipline of Software Engineering*. Reading, MA: Addison-Wesley, 1995.
24. Sims-Knight, J. E., & Upchurch, R. L., "Teaching object-oriented design to nonprogrammers: A progress report," *Proceedings of OOPSLA-92 Educators' Symposium*. Vancouver, British Columbia, Canada, 1992.
25. Sims-Knight, J. E. and R. L. Upchurch, "Teaching Object-Oriented Design Without Programming: A Progress Report," *Computer Science Education, 4*, 1992, pp. 135-156.
26. Collins, A., Brown, J. S., and Holum, A., "Cognitive apprenticeship: Making thinking visible," *American Educator*, Winter 1991, pp. 6-11, 38-46.
27. Upchurch, R., & Sims-Knight, J. E. "In Support of Student Process Improvement," Proceedings of CSEE&T'98, Atlanta, GA, February 22-25, 1998.
28. Anderson, J. R., Cornrad, F., & Corbett, A., "Skill acquisition and the Lisp Tutor," *Cognitive Science*, 13, 1989, pp. 467-505.
29. Pirolli, P., "Effects of examples and their explanation in a lesson on recursion: A production system analysis," *Cognition and Instruction*, 8, 1991, pp. 207-259.
30. Ratcliff, B. & Siddiqi, J., "An empirical investigation into problem decomposition strategies used in program design," *International Journal of Man-Machine Studies*, 22, 1984, pp. 77-90.
31. Berardi-Colletta, B., Buyer, L. S., Dominowski, R. L., & Rellinger, E., R. "Metacognition and problem solving: A process-oriented approach," *Jounral of Experimental Psychology: Learning, Memory, and Cognition*, 21, 1995, pp. 205-221.
32. Pirolli, P. & Recker, M., "Learning Strategies and Transfer in the Domain of Programming," *Cognition and Instruction*, 12, 1994.
33. Chi, M., de Leeuw, N., Chiu, M., & LaVancher, C., "Eliciting self-explanations improves understanding," *Cognitive Science*, 18, 1994, pp. 439-477.
34. Davidson, J. E., Deuser, R. & Sternberg, R. J., "The Role of Metacognition in Problem Solving," J. Metcalfe & A. P. Shimamura (eds.), *Metacognition: knowing about knowing*. Cambridge, MA: MIT Press, 1994, pp. 207-226.
35. Hsia, P. ,"Learning to Put Lessons Into Practice," *IEEE Software*, September 1993, pp. 14-17.
36. Cockburn, A., *Writing Effective Use Cases*. Reading, Massachusetts, Addison Wesley, 2001.
37. Fowler, M., *UML Distilled*. Reading, Massachusetts, Addison Wesley, 1997.
38. Bellin, D. & Simone, S., *The CRC Card Book*. Reading, Massachusetts, 1997.
39. Humphrey, W., *Introduction to the Team Software Process*. Reading, Massachusetts, Addison Wesley, 2000.
40. Williams, L. & Kessler, R., "Experimenting with Industry's "Pair-Programming" Model in the Computer Science Classroom," *Journal of Computer Science Education*, March 2001.
41. Williams, L. & Upchurch, R., "In Support of Student Pair-Programming," Technical Symposium on Computer Science Education, Charlotte, NC, February 21-25, 2001.
42. Turns, J. & Kurlik, A., "Structural Assessment to Support Engineering Education," ASEE, 1998.