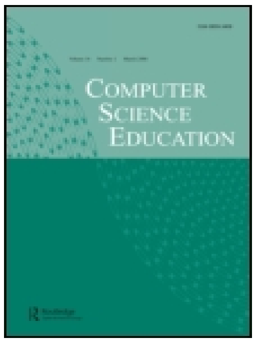

Computer Science Education

ISSN: 0899-3408 (Print) 1744-5175 (Online) Journal homepage: http://www.tandfonline.com/loi/ncse20

A survey of computer science capstone course literature

Robert F. Dugan Jr

To cite this article: Robert F. Dugan Jr (2011) A survey of computer science capstone course literature, Computer Science Education, 21:3, 201-267, DOI: 10.1080/08993408.2011.606118

To link to this article: http://dx.doi.org/10.1080/08993408.2011.606118

Published online: 15 Sep 2011.

Computer Science Education
Vol. 21, No. 3, September 2011, 201–267

SURVEY

A survey of computer science capstone course literature

Robert F. Dugan, Jr*

Department of Computer Science, Stonehill College, Easton, Massachusetts 02357, USA

(*Received 1 December 2009; final version received 14 July 2011*)

In this article, we surveyed literature related to undergraduate computer science capstone courses. The survey was organized around course and project issues. Course issues included: course models, learning theories, course goals, course topics, student evaluation, and course evaluation. Project issues included: software process models, software process phases, project type, documentation, tools, groups, and instructor administration. We reflected on these issues and the computer science capstone course we have taught for seven years. The survey summarized, organized, and synthesized the literature to provide a referenced resource for computer science instructors and researchers interested in computer science capstone courses.

Keywords: computer science education; capstone; software engineering; literature survey

1. Introduction

"Capstone courses generally target undergraduate students who are nearing completion of their studies. They are designed to build on skills acquired in earlier courses and emphasize situations and challenges that exist in the real world. Specific learning goals and course objectives vary across disciplines and institutions but most capstone courses provide an opportunity for students to demonstrate a range of professional competencies and communication skills." Rhodus and Hoskins (1995) cited by Stansfield (2005). Generally speaking, a computer science undergraduate capstone course consists of a final year, semester long, group-based, software engineering project. Since Parnas (1972) discussed a software engineering course for final year undergraduates in 1972, hundreds of papers have been written about this type of course but a comprehensive literature survey did not exist.

This article is a comprehensive ($\sim$200 papers) survey that organizes, summarizes, and synthesizes undergraduate computer science capstone literature to serve as a resource for:

*Email: bdugan@stonehill.edu

ISSN 0899-3408 print/ISSN 1744-5175 online
© 2011 Taylor & Francis
DOI: 10.1080/08993408.2011.606118
http://www.informaworld.com

- computer science instructors wishing to develop or enhance a capstone course
- researchers wishing to explore computer science capstone course research.

The survey is organized into two major sections. The first section discusses course issues including: course models, learning theories, course goals, course topics, student evaluation, and course evaluation. The second section discusses project issues including: software process models, software process phases, project type (industrial, community service, game, simulated, research), documentation, tools, groups (selection, size, organization), and instructor administration.

The survey also includes reflections on course issues, project issues, and the computer science capstone course we taught for seven years.

The major literature sources used in this survey were:

- The American Society of Engineering Education Journal of Engineering Education
- The Association of Computing Machinery Special Interest Group on Software Engineering Conference Proceedings
- The Journal of Computer Science Education
- The Journal of Computing Sciences in Colleges
- The Software Engineering Body of Knowledge
- The Software Engineering Education Knowledge
- The Software Engineering Institute Software Engineering Education Conference Proceedings.

Candidate papers were discovered by searching the title and abstract of the literature sources with the keywords "capstone" and "software engineering course". Candidate papers were eliminated if the abstract indicated that the paper did not focus on an undergraduate capstone or an undergraduate final year software engineering course. As candidate papers were examined, a pattern of common topics emerged. We categorized papers by topic and identified exemplary papers for each topic. If a candidate paper cited work outside the literature sources and the work covered a topic in an exemplary fashion, then we added the work to the survey. Several exemplary papers were also added to the survey based on reviewer suggestions.

2. Prior work

Most early surveys of software engineering courses sampled instructors rather than the nascent literature. In the 1970s and 1980s a common theme was lack of resources to teach the course: unqualified instructors, lack of extra preparation time needed for the course, inadequate textbooks, competition from other department sub-disciplines, and lack of time to combine theory and practice into a single semester course (Hoffman, 1978; Mynatt & Leventhal, 1987; Petricig & Freeman, 1984; Werth, 1987).

In the late 1980s as software engineering was maturing as a discipline, so were software engineering course textbooks. Carver (1987) looked at five popular software engineering textbooks to determine the course topics for a one-semester software engineering course:

System/software planning	Estimation/cost techniques	Software management
Software requirements/analysis	Software design process	User interface design
Software measurement	Programming methodology	Modularization
Design tools/representations	Programming languages	Verification/validation
Software reliability	Software maintenance	Documentation
Software psychology		

The formation of the Software Engineering Institute (SEI) at Carnegie Mellon accelerated the maturity of the software engineering discipline. During the 1990s the SEI reviewed US software engineering programs and software engineering literature and published several reports critiquing these programs to establish an ideal curriculum for software engineering (Bagert, 1998; Ford, 1991, 1994; Modesitt, Bagert, & Werth, 2001). The mature breadth and depth of these reports provided a wealth of ideas, but also made it challenging for an instructor trying to decide what to teach in a single semester capstone course.

In the mid-1990s, a full decade after the American Association of Colleges described the need for capstone course in all disciplines, computer science literature began to use the term "capstone course" to describe the undergraduate, final year, single semester, capstone course with some frequency (Hogan and Thomas, 2005). Todd, Magleby, Sorensen, Swan and Anthony (1995) surveyed several hundred engineering departments regarding capstone course description, project deliverables, groups, project sponsors, and faculty involvement. Dutson, Todd, Magleby, and Sorensen (1997) followed with a survey of engineering capstone course literature that had a deeper focus on groups and course evaluation. A key difference between our survey and Dutson et al.'s (1997) was that our survey focused exclusively on computer science capstone courses rather than all engineering capstone courses. Nevertheless we found the survey helpful when constructing the framework for our survey.

Fincher, Petre, and Clark (2001) published a book on computer science projects. The authors surveyed over 50 colleges and universities and analyzed the survey results. In the first part of the book, the authors catalogued the practice of computer science projects with common project types (e.g. instructor-based, client-based, research-based), common project tasks (e.g. creating groups, writing reports), and exemplar case studies. In the second part of the book, the authors gave advice in six project areas: allocating students and instructors to projects, supervising students, evaluating students, reflecting by students and instructors, managing groups, and motivating students. The book covered computer science project work in-depth and in an organized manner. A key difference between our survey and the book was that the book surveyed instructors, while we surveyed the literature.

Clear, Goldweber, Young, Leidig, and Scott (2001) and the Conference on Innovation and Technology in Computer Science Education formed a working group to discuss resources for instructors of undergraduate computer science capstone courses. The working group's report examined many of the same project oriented issues found in Dutson et al. (1997) and Fincher et al. (2001) summarized in this list:

- course goals
- characteristics of projects
- project deliverables
- sponsors
- groups
- prerequisites and preparation
- grading and assessment
- administration and supervision
- reflection, analysis and review.

The report consisted mainly of lists of issues to consider for a capstone course. For example, the course goals issue listed eight possible course goals as bulleted items. A list of references which included many course websites was provided at the end of the report, but the references were not cited within the report. The report also encouraged the reader to seek out a website which the authors intended to serve as a repository for capstone course best practices; however, this website appears to be inactive. Despite shortcomings, we found this report invaluable in our survey, particularly in the areas of course goals and instructor administration of the course project.

Bridgeman (2008) used the work of Clear et al. (2001) to survey 35 capstone papers and posters published at the Australian National Advisory Committee on Computing Qualifications conference between 1998 and 2007. Each work was classified according to which of Clear et al.'s (2001) nine questions were addressed. The authors found that many papers addressed course goals and deliverables, while few papers addressed groups and prerequisites/preparation.

In 2003, the final draft of the Software Engineering Body of Knowledge (SWEBOK) was published. SWEBOK organized software engineering literature into 10 knowledge areas: software design, software configuration management, software construction, software engineering infrastructure, software engineering management, software engineering process, software evolution and maintenance, software quality analysis, software requirements analysis, and software testing. Another objective was to differentiate software engineering from other disciplines such as computer science (Abran, Moore, & Bourque, 2004). SWEBOK was followed the next year by the Software Engineering Education Knowledge guide (SEEK). SEEK organized SWEBOK for software engineering education with detailed outline of a comprehensive software engineering curriculum based on SWEBOK's ten knowledge areas (Sobel, 2003). Both SEEK and SWEBOK provided a wealth of software engineering information, but their breadth and depth may be overwhelming to a computer science capstone instructor.

Surveys dating back to the 1970s have demonstrated academic interest in the computer science capstone course. To manage the many dimensions of the capstone course, authors developed frameworks around course and project issues. Trade-offs among the breadth of software engineering topics, the authentic project experience, and the limitations of a single semester course were often examined. With the exception of two limited surveys (Bridgeman, 2008; Mynatt & Leventhal, 1987) none of the surveys examined computer science capstone literature specifically and our survey addressed this shortcoming. Several small-scale surveys of computer science capstone course instructors have been conducted (Fincher et al., 2001; Streib & White, 2002), but a large-scale survey similar to Todd et al. (1995) needs to be conducted to understand how capstone courses are actually being taught.

3. Course: models

A course model was a framework upon which the instructor structured the capstone course. A common course model was a semester long course that covered software engineering and required students to complete a large software engineering project (Banios, 1992; Beasley, 2003; Gehrke et al., 2002; Hain & Jodis, 1992; Northrop, 1989).

An enhancement to this course model was to expand the semester long course into multiple semesters. One variant of this course model taught students the process of software engineering in one semester and allowed the students to focus on applying that process to a large project in the second semester (Adams et al., 2003; Davis, 2001; Goold, 2003; Hain & Jodis, 1992; Hamilton & Ruocco, 1998; Roggio, 2006; Tan & Phillips, 2005; Vaughn Jr, 2001). Another variant of this course model broke the software engineering process across multiple semesters, with the requirements and design phases taught in the first semester, and the implementation, testing and maintenance phases taught in the second semester (Bolz & Jones, 1983; Bruhn & Camp, 2004; Burge, 2007; Clear, 2007; Hadfield & Jensen, 2007). Gary et al. (2005) described a four-semester course model. In the first two semesters, juniors learned about tools, development, testing, and deployment, and implemented a system based on a design specification from seniors. In the second two semesters, seniors learned about requirements, analysis, project management, and design specification and managed juniors during the implementation.

Tvedt, Tesoriero, and Gary (2002) proposed an eight-semester course model where students across all years worked together on a large software project. Students were exposed to phases of the software lifecycle in reverse order. Integrating the model into an existing curriculum was challenging. The authors chose to do the integration all at once, but they also considered an incremental approach.

Another course model was a course that covered a research experience. Schneider (2002) discussed several reasons why a research experience capstone course would be superior to a software engineering experience capstone course. A research experience would be beneficial to all students, not just the best students. A single semester of software engineering was too short a period of time for students to undertake a large software project. Finally, software engineering capstones lacked a focus on written and oral communication.

Four major steps were involved in a research experience capstone course: (1) students selected a research topic which generally originated from one of their advanced classes, (2) students participated in a senior capstone seminar where they learned how to do research, (3) students wrote a capstone paper (up to three drafts) which had to be reviewed by an adviser and a second reader, (4) students gave a twenty minute talk about their research as part of an all day and capstone conference event. Students listed several benefits of a research experience capstone course: they gained self-confidence from completing a large research project, they received the chance to work closely with a single faculty member, they had the ability to show capstone paper to potential employers, and they had the chance to overcome fear of speaking to a large audience.

Huang (2003) made the case for using the game of Go as a capstone research experience. Algorithms to play the game of Go were an open research problem that was easy to understand but difficult to solve, and researchers in this area were happy to collaborate with students. Beasley, Cupp, Sanders, and Walker (2004) discussed a research experience capstone course that paralleled an advanced topic course. For example, an artificial intelligence research paper that was written in conjunction with an artificial intelligence course. A hybrid course model combined both the software engineering experience and the research experience (Adams et al., 2003).

Another course model was a course that covered a special topic. Grable (2003) integrated students' prior knowledge through a survey course in information theory. The author viewed a core idea in computer science as state change and presented a detailed syllabus for 14 different information theory topics. Sobel (1996) and Chiang (2004) devoted a significant portion of the course to the subject of formal methods in software engineering. Parberry, Roden, and Kazemzadeh (2005) described a two semester game programming course. In the first semester, the students learned the tools and art of game programming by working on small projects. In the second semester, students worked on a single game of their choosing. At the end of the second semester, students participated in a contest with judges from the game industry.

Many students in our capstone course suggested splitting the course across two semesters with requirements, specification, and design covered in the first semester and implementation, testing, and maintenance covered in the second semester. In the literature we saw variants of this from a single semester capstone to an eight semester capstone. While an eight-semester capstone provided instructors with time to cover software engineering principles in depth, we wondered whether the proponents of this approach were really advocating for a software engineering curriculum.

Both research experience and special topic capstone courses offered benefits not found in a conventional capstone course, but we believe that these courses have not mainstreamed because they lack the real world experience needed by the majority of final year students preparing for jobs in industry.

4. Course: learning theories

A learning theory described the learning framework that underpinned the capstone course. Learning theories were organized into four major types: behaviorism, cognitivism, constructivism, and humanism (Leonard, 1979).

Behaviorism viewed learning as new behavior triggered by the environment through classical and operant conditioning. Classical conditioning used stimulus to change behavior (e.g. positive feedback during class discussions lead to the expectation that future class discussions would also receive positive feedback). Operant conditioning used reward/punishment to change behavior (e.g. receiving an ''A'' for good project work). Behaviorism did not account for an individual's brain and personal experiences, and the theory waned in favor of cognitivism.

Cognitivism viewed learning as a new behavior triggered by brain activity rather than the environment. The brain activity used prior knowledge to process, organize, and store new knowledge. The brain was viewed like a computer. Sensory registers collected information from the senses. Short-term memory extracted important information from the sensory registers. Long-term memory and storage extracted important information from short-term memory and with reinforcement stored the information indefinitely. Once the information was stored, the brain found standard connections, called schema, between old and new information. Contrasted with behaviorism, cognitivism focused on the internal processing that created schemas between old and new information, while behaviorism was concerned with the external change in behavior change that occurred in response to the new information.

Constructivism viewed learning as the process of assimilating and accommodating new experiences. If a new experience did not contradict the brain's framework of understanding, then the experience was assimilated into the framework. If a new experience contradicted the brain's framework, then the framework was altered to accommodate the experience. Components of constructivism learning included: knowledge construction, skill learning, authentic tasks, collaboration, and related experience. Constructivism compared with cognitivism because both focused on the brain's internal processing, but contrasted with cognitivism because constructivism viewed the internal processing as unique to each learner. Piaget, who believed in learning by doing, was an early contributor to this learning theory. Virtually all of the learning theories discussed in the capstone course literature were some form of constructivism.

Humanism viewed learning as the process of self actualization. The instructor's role was to help students grow and mature through accepting ambiguity, accepting self, accepting others, self-conceptualizing, making personal decisions, and becoming a self-learner. The theory was rooted in the philosophies of Confucius and Aristotle. Superficially, constructivism and humanism were similar because they were concerned with the subjective construction of knowledge within the learner. However constructivism was concerned the act of knowledge construction, while humanism was concerned with the growth and maturation of the learner.

Hadjerrouit (2005) revised a capstone course using constructivist learning. Knowledge construction helped students learn about the process of software engineering. Skill learning helped students learn important software engineering skills such as performing software lifecycle tasks, writing documentation, communicating, and critical thinking. Authentic tasks helped students learn by motivating them through a real world software project. Collaboration helped students learn important groupwork skills, and helped students learn from each other. Related experience helped students learn from the good and bad decisions made on prior class projects. Anecdotally, students reported improved problem solving skills and a deepened understanding of software engineering principles compared to capstone course students from prior years. Students and the instructor also reported spending much more time on the course compared to prior years.

Problem-based learning was a form of constructivism that used ill-structured and complex problems, student-centered learning, coach/facilitator instructors, small groups, and self/peer assessment to help students learn (Brazier, Garcia, & Vaca, 2007). Hogan and Thomas (2005) described a problem-based learning framework for project-based courses: note initial reaction, analyze problem, activate prior knowledge, formulate learning objective, research learning objectives, report back, analyze additional issues, and wrap up. Use of the framework in first and second year courses improved student self-direction and group work in the final year capstone course. Heckendorn (2002) argued that a problem-based learning approach in a capstone course should be called project-based learning because the problem was larger, more complex, more applied, and process-oriented. A problem-based learning variant subjected students to dirty tricks such as conflicting requirements, creeping features, changing groups, and crashing hardware (Bridgeman, 2008; Dawson, 2000). Bridgeman (2008) changed groups after the design document was completed. Although the overall course evaluation did not change from prior years (without dirty tricks), student comments indicated that they learned about resiliency and the importance of good documentation.

Active learning, another form of constructivism, achieved learning through a student-centered desire to discover, experience, and understand. Ludi, Natarajan, and Reichlmayr (2005) described a basic lecture/project course model that was redesigned to use active learning. The redesigned course consisted of 1/3 lecture in which students had to prepare for in advance, 1/3 class experience to reinforce lecture, and 1/3 project work. The redesigned course gave the students a better appreciation of software engineering principles, and resulted in more project successes compared to the original lecture/project based course.

In contrast to problem-based learning and active learning, Knight and Horton (2005) advocated the use of the studio-based learning theory. This theory emerged from the architecture discipline where students worked in small groups on a well defined problem (AACU, 1990). Each week, a subset of the groups presented to the rest of the class and received feedback from the instructor and other groups. All groups learned something from this feedback. Students were motivated by presentations and by a contest at the end of the semester. A studio-based learning capstone course scaled better (one instructor, six teaching assistants, 130 students) than a lecture/project capstone course (Knight and Horton, 2005).

Lynch, Goold, and Blain (2004b) used a survey to compare student preferences in a studio-based learning capstone course, a project/lecture capstone course with light instructor involvement, and a project/lecture capstone course with strong instructor involvement. Regardless of the approach, students preferred clearly defined project objectives, a clearly defined project schedule, and instructor involvement. Students with heavy instructor involvement viewed individual contributions as significant and believed that the capstone prepared them for an industry career, while students in the other courses did not share these views. The results of the study were not surprising and it would be interesting to conduct a future study that compares student learning, rather than student preferences across these three course styles.

Clark and Boyle (1999) reviewed four theories of teaching and learning based on the work of Fox (1983): transferring, shaping, growing, and traveling. In the transferring model, simplistic learning occurred through the acquisition of knowledge during lecture. In the shaping model, deeper learning occurred through lecture and exercises that reinforced the lecture. In the growing model, still deeper learning occurred through activities outside lecture. The instructor had much less control over these activities and student exposure to discipline-specific content. In the traveling model, the deepest learning occurred through teacher guided activities outside lecture. Like the growing model, the student played an important role in the learning process. Unlike the growing model, the instructor's guidance exposed the student to discipline-specific content in a controlled manner. The authors advocated using the traveling model because of its affinity with enculturation. Enculturation was the process of learning about a discipline through activity, concepts, and culture. The authors argued that the only way to truly understand a science discipline like computer science was through a process of enculturation in a final year project.

Dubinsky and Hazzan (2005) developed a theory of teaching and learning based specifically on the capstone course. The theory consisted of a set of ten principles (course teaching: focus on project, adjust software process to fit course, align tools with software process, connect software process with real world; people: evaluate student's use of software process, listen to student's successes and failures with software process, require student reflection; listen to mentor's successes and failures with software process; meta: inspire use of the software process rather than preaching process use) and five practices (release three iterations of project; assign clear group roles; measure task time, communication, and role performance; get feedback from mentors; evaluate student based on group

(65%) and role (35%) performance). The theory emerged over a period of five years, 300 students, and 31 capstone projects using a qualitative active research process to collect course data, analyze course data, and refine the theory.

Rather than modify the capstone course to fit a learning theory, Hauer and Daniels (2008) looked at learning theories that fit the capstone course. The authors situated the capstone course with in a broader category of Open Ended Group Projects (OEGPs). At its core an OEGP was an ill structured problem that was solved by a group. An OEGP had many benefits, but the authors cautioned that without instructor scaffolding the high cognitive demands placed on inexperienced students would result in a focus on product rather than process. Many learning theories applied to OEGPs: Situated Learning, Practice Fields, Communities of Practice, Bounded Rationality, Vygotsky's Zone of Proximal Development, and Cognitive Load. The authors argued that two theories that were most applicable were Holistic Theory and Activity Theory. Holistic Theory, a form of humanism, viewed learning as the process of human development through interconnecting three domains: cognitive (systematizes knowledge), perceptual (interprets interactions with the world), and affective (human values like ethics). For OEGPs the cognitive domain focuses on process, the perceptual domain focused on group interaction, and the affective domain focuses on ethics. Hauer and Daniels (2008) found that the affective domain was neglected by other learning theories. Activity Theory, a form of constructivism, viewed learning as the process of learning as activity through the Activity Triangle. The Triangle consisted of six interconnected domains: subjects, tools, object, division of labour, community, and rules (Mwanza, 2001). For OEGPs, Activity Theory was student-centered where subjects (the students) interacted with objects (the project artifacts) indirectly through the other Activity Triangle domains.

Given the volume of literature about computer science capstone courses, it was surprising that so little work was based on formal learning theory. The work that did discuss learning theory and the capstone course was reflective with weak evidence to support the efficacy of the learning theory. Two exceptions were Dubinsky and Hazzan (2005) and Hauer and Daniels (2008). The learning theory work was also dominated by the constructivist learning theory which leaves three major learning theories unexamined with respect to the capstone course.

Originally our capstone course used an in-class lecture/out-of-class project teaching style which we loosely classified as Fox's transferring model with poorly executed problem-based learning. The instructor was overly involved in the lecture component of the course (which the students found uninteresting), and the instructor was under involved in the project component of the course (which the students found interesting, but overwhelming). Over time the course improved through a mix of constructivist learning styles. Each project was large, ill-structured, complex, and authentic. Each week students used just-in-time learning by reading and reporting to the class on a software engineering process topic. The topic linked to the project's current software engineering phase. For example, a week before meeting with the client to gather requirements, the students read and reported on different requirements processes. Each week the students and the instructor also met to discuss the project. The instructor acted as a mentor to help the students link software engineering theory to practice and to share past project experience. Anecdotally we observed increased interest in acquiring timely software engineering knowledge linked to the project, a deepened sense of project ownership, and more project success.

5. Course: goals

As early as 1972, Parnas (1972) discussed course goals for a single semester software engineering course: teaching students how to work on a large group project, teaching students how to read and digest software engineering literature, and teaching students problem-solving techniques. The author made a particular point about teaching problem-solving techniques rather than teaching about current software engineering tools.

We created a taxonomy of course goals organized into three main areas: curriculum guidelines, discipline skills, and learning skills. The taxonomy was derived by collecting and organizing course goals from the literature.

Curriculum guidelines were capstone course goals described by various accreditation, academic, and professional organizations. The United States Accreditation Board for Engineering and Technology (ABET) criteria for accrediting computing programs was frequently cited in the literature with a particular focus on Criterion 3: Program Outcomes (ABET, 2005; Banios, 1992; Christensen & Rundus, 2003; Dutson et al., 1997; Ford, 1991; Kishline et al., 1998; Kishline, Wang, & Aggoune, 1998; Lancor, 2008; Miller & Olds, 1994; Neumann & Woodfill, 1998; Pilskalns, 2009; Reichlmay, 2006; Todd et al., 1995). The ACM/IEEE curriculum guidelines for computer science and software engineering were also frequently mentioned with a focus on skills needed for large scale software development (Akingbehin, Maxima, & Tsuia, 1994; Beasley et al., 2004; Beck, 2005; Budgen & Tomayko, 2003; Carver, 1987; Catanio, 2006; Leidig, Ferguson, & Leidig, 2006; Leslie, Waguespack, & Hass, 1984; Meer & Sigwart, 1985; Su, Jodis, & Zhang 2007; Way, 2005). The Association of American Colleges recommended that a capstone course in any discipline should provide the opportunity for knowledge integration, reflection, and critical thinking (AACU, 1990).

Discipline and learning skills are listed in Table 1.

Like course topics, there were many capstone course goals that competed for the instructor's attention. The top 10 discipline skills (by

Table 1. Course goals: discipline and learning skills.

Skill category	Skill
Discipline skills	– general methodology (Banios, 1992; Bernhart, Grechenig, Hetzl, & Zuser, 2006; Brazier et al., 2007; Buchheit, Ruocco, & Welch, 1999; Chiang, 2004; Cordes & Parrish, 1993; Ford, 1982; Freeman, Wasserman, & Fairley, 1976; Hamilton & Ruocco, 1998; Monroe & Yu, 1998; Monroe & Yu, 1998; Owen, 1989; Parrish et al., 1998; Sobel, 1996; Sun & Decker, 2004)
Software process	– specific methodology (Catanio, 2006; Shumba, 2005)
	– justification (Bolz & Jones, 1983; Catanio, 2006; Perkins & Beck, 1980)
	– requirements (Banios, 1992; Bolz & Jones, 1983; Brazier et al., 2007; Bruhn & Camp, 2004; Burge, 2007; Capon, 1999; Carver, 1987; Catanio, 2006; Chiang, 2004; Christensen & Rundus, 2003; Christensen & Rundus, 2003; Cordes & Parrish, 1993; Freeman et al., 1976; Gehrke et al., 2002; Gehrke et al., 2002; Hadfield & Jensen, 2007; Ludi et al., 2005; Miller & Olds, 1994; Parnas, 1972; Perkins & Beck, 1980; Perkins & Beck, 1980; Sobel, 1996)
	– design (Bolz & Jones, 1983; Brazier et al., 2007; Bruhn & Camp, 2004; Capon, 1999; Carver, 1987; Catanio, 2006; Chiang, 2004; Christensen & Rundus, 2003; Collofello & Woodfield, 1982; Cordes & Parrish, 1993; Freeman et al., 1976; Gehrke et al., 2002; Gehrke et al., 2002; Hadfield & Jensen, 2007; Henry, 1983; Kishline et al., 1998; Leidig et al., 2006; Ludi et al., 2005; Parnas, 1972; Parrish et al., 1998; Perkins & Beck, 1980; Streib & White, 2002; Wallace & Crow, 1995)
	– implementation (Bolz & Jones, 1983; Capon, 1999; Catanio, 2006; Chiang, 2004; Christensen & Rundus, 2003; Clear, 2007; Cordes & Parrish, 1993; Freeman et al., 1976; Gehrke et al., 2002; Henry, 1983; Perkins & Beck, 1980)
	– coding standards (Hadfield & Jensen, 2007)
	– testing (Bruhn & Camp, 2004; Carver, 1987; Catanio, 2006; Chiang, 2004; Christensen & Rundus, 2003; Christensen & Rundus, 2003; Clements, 2007; Collofello & Woodfield, 1982; Cordes & Parrish, 1993; Freeman et al., 1976; Gehrke et al., 2002; Gehrke et al., 2002; Hadfield & Jensen, 2007; Henry, 1983; Ludi et al., 2005; Parnas, 1972; Perkins & Beck, 1980; Rising, 1989; Sobel, 1996)
	– maintenance/reengineering/evolution (Bernhart et al., 2006; Brazier et al., 2007; Carver, 1987; Catanio, 2006; Chiang, 2004; Collofello & Woodfield, 1982; Gehrke et al., 2002; Hadfield & Jensen, 2007; Rising, 1989)
	– change management (Hadfield & Jensen, 2007)
	– industry standards (Hadfield & Jensen, 2007)
	– prototyping (Banios, 1992; Hadfield & Jensen, 2007; Ludi et al., 2005; Monroe & Yu, 1998; Perkins & Beck, 1980)
	– tools (Beck, 2005; Bernhart et al., 2006; Carver, 1987; Ford, 1982; Gehrke et al., 2002; Hadfield & Jensen, 2007; Kishline et al., 1998; Monroe & Yu, 1998; Parnas, 1972; Rising, 1989)

(*continued*)

Table 1. (*Continued*).

Skill category	Skill
	– integration (Catanio, 2006; Gehrke et al., 2002)
	– artifacts (Christensen & Rundus, 2003; Freeman et al., 1976)
	– formal methods (Henry, 1983; Sobel, 1996)
Discipline skills	– project management (Banios, 1992; Buchheit et al., 1999; Carver, 1987; Catanio, 2006; Chiang, 2004; Christensen & Rundus, 2003; Collofello & Woodfield, 1982; Cordes & Parrish, 1993; Freeman et al., 1976; Gehrke et al., 2002; Monroe & Yu, 1998; Perkins & Beck, 1980; Sobel, 1996)
Project management	– groupwork (Banios, 1992; Bareiss & Griss, 2008; Beck, 2005; Bolz & Jones, 1983; Burge, 2007; Capon, 1999; Chiang, 2004; Christensen & Rundus, 2003; Collofello & Woodfield, 1982; Cordes & Parrish, 1993; Distasio & Way, 2007; Ford, 1982; Gehrke et al., 2002; Henry, 1983; Hogan & Thomas, 2005; Kishline et al., 1998; Perkins & Beck, 1980; Perkins & Beck, 1980; Polack-Wahl, 1999; Ramakrishan, 2003; Sharon, 2001; Streib & White, 2002; Wallace & Crow, 1995)
	– conflict resolution (Goold, 2003)
	– negotiation (Capon, 1999; Goold, 2003)
	– giving/receiving feedback (Ecker, Caudill, Hoctor, & Meyer, 2004)
	– planning (Bareiss & Griss, 2008; Freeman et al., 1976; Gehrke et al., 2002; Goold, 2003; Neumann & Woodfill, 1998)
	– time management (Goold, 2003)
	– speaking (Banios, 1992; Bruhn & Camp, 2004; Capon, 1999; Christensen & Rundus, 2003; Ecker et al., 2004; Gehrke et al., 2002; Henry, 1983; Miller & Olds, 1994; Miller & Olds, 1994; Neumann & Woodfill, 1998; Perkins & Beck, 1980; Schneider, 2002; Stansfield, 2005; Towhidnejad & Aman, 1996)
	– reading (Brazier et al., 2007; Christensen & Rundus, 2003; Kishline et al., 1998)
	– writing (Brazier et al., 2007; Bruhn & Camp, 2004; Capon, 1999; Carver, 1987; Christensen & Rundus, 2003; Clements, 2007; Ecker et al., 2004; Gehrke et al., 2002; Kishline et al., 1998; Miller & Olds, 1994; Owen, 1989; Perkins & Beck, 1980; Pokorny, 2005; Schneider, 2002; Stansfield, 2005; Towhidnejad & Aman, 1996)
	– client management (Miller & Olds, 1994)
Discipline skills	– professional skills (Bareiss & Griss, 2008)
	– corporate environment (Bolz & Jones, 1983; Clear et al., 2001; Towhidnejad, 2002)
	– ethics (De Rego et al., 2005; Fisher & Abunawass, 1994; Kishline et al., 1998)
Other	– specialized topic (Stansfield, 2005)
	– self-awareness (strengths/weaknesses/mistakes) (Ramakrishan, 2003; Bosnić & Orlić, 2008)
	– self-confidence (Schneider, 2002)
	– good citizenship (Sun & Decker, 2004)
	– mentoring (Schneider, 2002)
	– multidisciplinary (Miller & Olds, 1994)
	– research (Capon, 1999; Clear et al., 2001; Long & Thoreson, 1986; Miller & Olds, 1994)
	– post-graduate (Clear et al., 2001)
	– large system (Banios, 1992; Bernhart et al., 2006; Bolz & Jones,

(*continued*)

Table 1. (*Continued*).

Skill category	Skill
	1983; Catanio, 2006; Christensen & Rundus, 2003; Clear et al., 2001; Gehrke et al., 2002; Hamilton & Ruocco, 1998; Henry, 1983; Miller & Olds, 1994; Miller & Olds, 1994; Owen, 1989; Sharon, 2001; Towhidnejad, 2002)
Learning skills	– knowledge integration (Banios, 1992; Christensen & Rundus, 2003; Clear et al., 2001; Clements, 2007; Ecker et al., 2004; Goold, 2003; Neumann & Woodfill, 1998; Pokorny, 2005; Sharon 2001; Stansfield, 2005)
	– reflection (Burge, 2007; Clear et al., 2001; Ecker et al., 2004; Neumann & Woodfill, 1998)
	– critical thinking (Burge, 2007; Burge, 2007; Kishline et al., 1998)
	– application (Bareiss & Griss, 2008; Burge, 2007; Kishline et al., 1998; Neumann & Woodfill, 1998)
	– lifelong learning (Parrish et al., 1998; Ramakrishan, 2003; Towhidnejad, 2002)

citation frequency) were: design (24), requirements (22), groupwork (21), testing (19), writing (17), speaking (15), software process (14), project management (14), large system experience (14) and knowledge integration (10). The top 10 skills accounted for 62% of the citations and the top five skills account for 41%. When we compared the top 10 goal list against the goals in our own capstone course, we found again that we needed more focus on the testing goal.

von Konsky, Hay, and Hart (2008) presented an alternative, exhaustive, and richer list of skills for entry level software engineers that used Australia's Skills Framework for the Information Age (SFIA). The SFIA organized a list of 78 business and software development skills for information technology professionals. The authors found that students graduating from the author's program had the skills identified by SFIA.

6. Course: topics

We collected course topics that appeared in the literature and organized them using the SEEK taxonomy (Sobel and LeBlanc, 2002) (see Table 2). In most cases, course topics were listed as part of syllabus with little detail, but there were some exceptions (Akingbehin et al., 1994; Banios, 1992; Bruhn & Camp, 2004; Carver, 1987; Christensen & Rundus, 2003; Collofello & Woodfield, 1982; Dubinsky & Hazzan, 2005; Monroe & Yu, 1998; Parnas, 1972; Perkins & Beck, 1980; Rising, 1989).

An alternative taxonomy was presented in Bernhart et al. (2006) which used SWEBOK knowledge areas and disciplines, and applied that taxonomy to two different software engineering courses. An additional resource was the 28 Software Engineering Institute curriculum modules

Table 2. Summary of course topics organized around the SEEK taxonomy.

SEEK area	Course topic
Fundamentals	– formal methods (Henry, 1983; Sobel, 1996) – software metrics (Carver, 1987; Henry, 1983)
Professional practice	– documentation (Brazier et al., 2007; Bruhn & Camp, 2004; Capon, 1999; Carver, 1987; Christensen & Rundus, 2003; Clements, 2007; Ecker et al., 2004; Gehrke et al., 2002; Kishline et al., 1998; Miller & Olds, 1994; Owen, 1989; Parrish et al., 1998; Perkins & Beck, 1980; Pokorny, 2005; Schneider, 2002; Stansfield, 2005) – software psychology (Carver, 1987) – groupwork (Banios, 1992; Bareiss & Griss, 2008; Beck, 2005; Bolz & Jones, 1983; Burge, 2007; Capon, 1999; Chiang, 2004; Christensen & Rundus, 2003; Collofello & Woodfield, 1982; Cordes & Parrish, 1993; Distasio & Way, 2007; Ford, 1982; Gehrke et al., 2002; Henry, 1983; Kishline et al., 1998; Perkins & Beck, 1980; Polack-Wahl, 1999; Ramakrishan, 2003; Sharon, 2001; Streib & White, 2002; Wallace & Crow, 1995) – ethics (De Rego, Zoltowski, Jamieson, & Oakes, 2005; Fisher & Abunawass, 1994; Kishline et al., 1998) – social/global impact of technology (De Rego et al., 2005) – presentation (Banios, 1992; Bruhn & Camp, 2004; Capon, 1999; Christensen & Rundus, 2003; Ecker et al., 2004; Gehrke et al., 2002; Henry, 1983; Miller & Olds, 1994; Neumann & Woodfill, 1998; Perkins & Beck, 1980; Schneider, 2002; Stansfield, 2005; Towhidnejad & Aman, 1996)
Software requirements	– requirements (Banios, 1992; Bolz & Jones, 1983; Brazier et al., 2007; Bruhn & Camp, 2004; Burge, 2007; Capon, 1999; Carver, 1987; Catanio, 2006; Chiang, 2004; Christensen & Rundus, 2003; Christensen & Rundus, 2003; Cordes & Parrish, 1993; Freeman et al., 1976; Gehrke et al., 2002; Hadfield & Jensen, 2007; Ludi et al., 2005; Miller & Olds, 1994; Parnas, 1972; Perkins & Beck, 1980; Sobel, 1996)
Software design	– design (Bolz & Jones, 1983; Brazier et al., 2007; Bruhn & Camp, 2004; Capon, 1999; Carver, 1987; Catanio, 2006; Chiang, 2004; Christensen & Rundus, 2003; Collofello & Woodfield, 1982; Cordes & Parrish, 1993; Freeman et al., 1976; Gehrke et al., 2002; Gehrke et al., 2002; Hadfield & Jensen, 2007; Henry, 1983; Kishline et al., 1998; Leidig et al., 2006; Ludi et al., 2005; Parnas, 1972; Parrish et al., 1998; Perkins & Beck, 1980; Streib & White, 2002; Wallace & Crow, 1995) – user interface (Carver, 1987; Ludi et al., 2005) – reverse engineering (Gehrke et al., 2002) – design languages (Carver, 1987; Henry, 1983; Ludi et al., 2005; Perkins & Beck, 1980) – reusability (Chiang, 2004) – portability (Chiang, 2004) – interoperability (Chiang, 2004)
Software construction	– prototyping (Banios, 1992; Monroe & Yu, 1998; Perkins & Beck, 1980; Ludi et al., 2005; Hadfield & Jensen, 2007) – implementation (Cordes & Parrish, 1993; Clear, 2007; Gehrke et al., 2002; Henry, 1983; Chiang, 2004; Perkins & Beck, 1980; Bolz & Jones, 1983; Christensen & Rundus, 2003; Freeman et al., 1976; Catanio, 2006; Capon, 1999)

(*continued*)

Table 2. (*Continued*).

SEEK area	Course topic
	– programming languages/methodology (Parnas, 1972; Carver, 1987; Chiang, 2004; Monroe & Yu, 1998)
	– tools (Monroe & Yu, 1998; Carver, 1987; Rising, 1989; Parnas, 1972; Hadfield & Jensen, 2007; Gehrke et al., 2002; Kishline et al., 1998; Beck, 2005; Bernhart et al., 2006; Ford, 1982)
Software process	– life-cycle models (Monroe & Yu, 1998; Chiang, 2004; Sobel, 1996; Monroe & Yu, 1998; Buchheit et al., 1999; Parrish et al., 1998; Cordes & Parrish, 1993; Sun & Decker, 2004; Bernhart et al., 2006; Ford, 1982; Banios, 1992; Brazier et al., 2007; Hamilton & Ruocco, 1998; Freeman et al., 1976; Owen, 1989)
Verification and validation	– test (Cordes & Parrish, 1993; Collofello & Woodfield, 1982; Parnas, 1972; Rising, 1989; Carver, 1987; Bruhn & Camp, 2004; Christensen & Rundus, 2003; Henry, 1983; Gehrke et al., 2002; Chiang, 2004; Perkins & Beck, 1980; Ludi et al., 2005; Sobel, 1996; Hadfield & Jensen, 2007; Gehrke et al., 2002; Clements, 2007; Christensen & Rundus, 2003; Freeman et al., 1976; Catanio, 2006)
	– design/code inspections (Rising, 1989; Ludi et al., 2005; Perkins & Beck, 1980; Gehrke et al., 2002; Henry, 1983)
Software evolution	– maintenance (Collofello & Woodfield, 1982; Rising, 1989; Carver, 1987; Chiang, 2004; Gehrke et al., 2002; Hadfield & Jensen, 2007; Gehrke et al., 2002; Brazier et al., 2007; Catanio, 2006; Bernhart et al., 2006)
Software quality	
Software management	– project management (Collofello & Woodfield, 1982; Carver, 1987; Banios, 1992; Christensen & Rundus, 2003; Chiang, 2004; Perkins & Beck, 1980; Sobel, 1996; Perkins & Beck, 1980; Gehrke et al., 2002; Monroe & Yu, 1998; Buchheit et al., 1999; Cordes & Parrish, 1993; Freeman et al., 1976; Catanio, 2006)
	– risk management (Banios, 1992)
Other	– introduction/overview (Banios, 1992; Henry, 1983; Bruhn & Camp, 2004; Monroe & Yu, 1998; Christensen & Rundus, 2003; Gehrke et al., 2002; Perkins & Beck, 1980; Ludi et al., 2005)
	– literature review (Parnas, 1972)
	– guest lectures (Christensen & Rundus, 2003)
	– exam (Christensen & Rundus, 2003; Banios, 1992; Henry, 1983)
	– project work (in class) (Christensen & Rundus, 2003)

covering the major areas of software engineering. Each curriculum module was intended to be used as standalone course topic (Budgen and Tomayko, 2003).

With so many software engineering topics, it could be challenging for instructor to decide which topics to include in a capstone course. One measure of topic importance was to look at the frequency of literature citations by topic. The top ten topics (by citation frequency) in Table 2 were in order: design (24), groupwork (22), requirements (20), test (18), documentation (17), lifecycle (15), presentation (15), project management

(13), maintenance (10), and tools (10). These topics account for 93% of the citations. The top five topics account for 54% of the citations.

When we compared the top 10 topic list against the topics we cover in our own capstone course, we found maintenance missing completely from our syllabus. We spent time talking about test in the course, but probably not enough time considering that the topic appeared in the top five topic list.

7. Course: student evaluation

A straightforward student evaluation was based on a combination of course artifacts (homework and exams) and project artifacts (documentation, code, final presentation, final report) (Fincher et al., 2001; Henry, 1983; Mynatt & Leventhal, 1987; Richards, 2009; Tan & Phillips, 2005). In an early survey of software engineering courses, Mynatt and Leventhal (1987) found that 60% of the student evaluation was based on written and oral reports and exams, with the rest of the student evaluation based on a course project. In later work Richards (2009) surveyed the literature on student evaluation in computer science project courses and developed a list of course goals against which students were evaluated:

- analyze, describe, and apply principles and models of software development.
- perform the stages of the software engineering lifecycle (requirements analysis, design, implementation, testing) in an authentic context.
- demonstrate an understanding of the influences of group effectiveness and strategies for supporting effective interoperation.
- demonstrate the capacity to work effectively in a software development group.
- effectively communicate results of the software development process in both written and oral form.

The author created a table of course and project artifacts; the course goals achieved by each artifact; whether the artifact was an individual effort, group effort, or both; and a grade weighting for the artifact.

While a small number of group projects were easier to evaluate than a large number of individual projects (DeClue, 2007), evaluating individual students within a group was challenging (Chamillard & Braun, 2002; Conn, 2004; Fincher et al., 2001; Lancor, 2008; Rising, 1989; Stein, 2003). A common observation was the wide variability in performance of students within the same group which made a single group grade unfair. One study found some students spent two to five times more hours on the group project than other students within the group (Stein, 2003).

Richards (2009) summarized Lejk and Wyvill (1996)'s approaches to evaluating individual students within the group:

- student distribution of grade points within a group (peer evaluation)
- peer evaluation of project tasks and group maintenance tasks (peer evaluation)
- multiplying the group grade by an individual weighting (instructor/self/peer/group evaluation)
- group grade plus/minus individual contribution grade (instructor/self/peer/group evaluation)
- shared group grade with instructor intervention where necessary (instructor evaluation)

Our survey organized evaluating students within a group based on the evaluator and artifact: instructor evaluation, self-evaluation, peer-evaluation, group leader evaluation, other participant evaluation, and select project artifact evaluation.

Self-evaluation, peer-evaluation, and group leader evaluation allowed the instructor to evaluate individual students within a group using the subjective input of group members. Although subjective, this type of evaluation generally supported instructor observations (Collofello, 1985). A short evaluation asked each student to grade other students in the group. The student's project grade was modified by the grade the student received from other students in the group (Chamillard & Braun, 2002; Henry, 1983; Swift & Neebel, 2003). Richards (2009) used an individual contribution form completed by each student for each project artifact. The form asked the student to list the percentage contribution of each member of the group for the artifact. The instructor used the forms to scale the group grade for each individual. A longer evaluation might include: the number of hours spent on the project; an analysis of each member's contribution to the project; a Likert scale rating of project management skills (time management, stress management, cooperation, responsiveness to others, decision-making, leadership, and self-confidence); a Likert scale rating of project contributions at each software process phase (AACU, 1990; Bergin, 2002; Fincher et al., 2001). Vaughn Jr (2001) found that peer-evaluations were more accurate later in the semester.

In addition to the instructor and students participating in the student evaluation, other participants may have included: co-instructors (Ecker et al., 2004; Fincher et al., 2001), project clients (Bruhn & Camp, 2004; Fincher et al., 2001; Judith et al., 2003; Vaughn Jr, 2001), and members of the instructor's department (Clear et al., 2001; Fincher et al., 2001). Parkers et al. (1999) warned that students found it difficult to refuse client requests for additional project features late in the software lifecycle if the client participated in the student evaluation. With so many student evaluation participants, and the qualitative nature of project evaluation, it was critical that theinstructorthe instructor provided clear expectations and transparency for student evaluation (Clear et al., 2001; Fincher et al., 2001).

Select project artifacts could be used to evaluate an individual student within a group. Project collaboration tools that included features such as task list management and messaging helped facilitate groupwork, but could also be useful when evaluating students on a group (Lancor, 2008). Version management system tools could measure the number of lines of code and the number of defects contributed by each student (Dick, Postema, & Miller, 2000; Gehrke et al., 2002). Activity logbooks, meeting minutes, and progress reports could record student activity and how many hours the student spent on that activity (Burge, 2007; Dick et al., 2000; Fincher et al., 2001; Henry, 1983; Lancor, 2008; Stein, 2003; Swift & Neebel, 2003). Attendance at group meetings could also be used (Tan & Phillips, 2005).

A student evaluation could simply be a grade computed from the points assigned to course and project artifacts (Chamillard & Braun, 2002; Henry, 1983; Mynatt & Leventhal, 1987; Swift & Neebel, 2003; Tan & Phillips, 2005), but alternatives were reported. Fincher et al. (2001) devised a grading taxonomy that included: establishing weighted/unweighted categories, negotiating categories and weighting, listing attributes of high quality and low quality projects, or some combination. In one course, the instructor wrote a three-page student evaluation that was added to the student's permanent transcript (Cushing, Cunningham, & Freeman 2003). In another course, the instructor rated each student on a scale of one to five against the ABET criteria (AACU, 1990; Kishline et al., 1998). Several papers reported on the success of using virtual money as a method for evaluating students (Bergin, 2002; Fincher et al., 2001; Stansfield, 2005). In one course, students were given a salary based on a position in a virtual company. The greater work and responsibility required for a position, the greater the salary. Each student received two merit views during the semester which could have resulted in salary increases (Clifton, 1991a). In another course, students were required to complete a project consisting of three modules. Starting with \$10,000 of virtual money, students were required to complete one module on their own and purchase the other modules from other students in the class. At the end of the semester the instructor purchased the student's project. The purchase price was based on the quality of the project artifacts (Mazlack, 1981).

Regardless of what student evaluation techniques were used, it was important that the students were evaluated and provided with feedback on a continuous basis during the capstone course. This allowed small problems to be detected and corrected early before they became large problems (Collofello, 1985; Fincher et al., 2001; Lancor, 2008).

In our capstone course we evaluated our students using course/project artifacts and informal feedback from students and clients. Everyone in the class worked on the same demanding project and we met weekly to discuss project progress. This weekly feedback allowed us to identify student participation problems early. After an intervention, if the student was unable to fully participate in the project, then the student was put in a low risk project role and the student's project grade was adjusted. The final grade was mostly dependent upon the project but we found that by the end of the course most of the students were motivated by pride and ownership in the project rather than the final grade.

8. Course: evaluation

There were many techniques for course evaluation including: reflection, surveys, focus groups, project success, and a variant of the capability maturity model. Instructor reflection, a qualitative course evaluation, appeared frequently in the literature (Anderson, 1987; Ecker et al., 2004; Fincher et al., 2001; Rising, 1989; Way, 2005). Common observations that are emerged from instructor reflection included:

- students had difficulty writing software engineering documentation (Bergin, 2002)
- real projects and clients made the learning experience unpredictable (Bergin, 2002)
- the balance between process and product was difficult to maintain (Bergin, 2002)
- students didn't communicate effectively within a group (Bickerstaff and Moro, 2006)
- all participants (student, instructor, client) benefited from the project (Hutchens and Katz, 1996)

Davis's (2001) student reflection was a technique for both course evaluation and student learning. The reflection examined how personalities, roles, and group dynamics impacted the project. The author was the group's leader, but because of his introverted personality he had trouble expressing his vision to the group. The lack of a coherent vision impacted the final project. Participation by group members was uneven. For example, while a well written software requirements document was completed on time by the group, the configuration management plan document was a disaster because this document relied on the efforts of a group member who was not pulling her own weight.

Surveys were a quantitative course evaluation technique. The survey source population was most often students in the course, but could also include course alumni (Northrop, 1989), alumni employers (Northrop, 1989), project clients (Burge, 2007; Tan & Phillips, 2005), academic colleagues (Sharon M., 2001), and industry colleagues (Way, 2005). The most common form of survey was a course evaluation completed at the end of the course. Survey questions included Likert scale questions like the following general Likert scale questions:

- Was the course as a whole good? (Northrop, 1989)
- Was the course content good? (Northrop, 1989)
- Was instructor's contribution to course good? (Northrop, 1989)
- Was the instructor's effectiveness as a teacher good? (Northrop, 1989)
- How was this course compared with other courses? (Northrop, 1989)
- Were the interest and level of class lectures good? (Northrop, 1989)
- How was the pace of the course? (Ludi et al., 2005)
- How was the amount of lecture? (Ludi et al., 2005)
- Were the course objectives helpful? (Ludi et al., 2005)
- Was the course outline helpful? (Ludi et al., 2005)
- Did you prepare for the lecture? (Ludi et al., 2005)

Project related Likert scale questions:

- Were analysis and design important skills? (Catanio, 2006)
- Was project management an important skill? (Catanio, 2006)
- Was the project enjoyable? (Gehrke et al., 2002)
- Was the project helpful? (Ludi et al., 2005)
- How would you evaluate the work you were responsible for? (Gehrke et al., 2002)
- How would you evaluate the groupwork of your group? (Gehrke et al., 2002)
- Did you learn something about project development? (Gehrke et al., 2002)
- How would you evaluate your use of software engineering techniques? (Gehrke et al., 2002)
- Did the software engineering course make sense to you? (Gehrke et al., 2002)
- How was the amount of class time for the project? (Ludi et al., 2005)
- How well did you prepare for the project presentations? (Ludi et al., 2005)

Alternatively, some of these Likert scale questions may have appeared as questions inviting written comments. Surveys could have also included space for additional written comments.

Unsolicited comments were yet another course evaluation technique, with the same source population as surveys (Knight & Horton, 2005; Way, 2005).

Chamillard and Braun (2002) reported on the use of a student management group and focus groups as course evaluation techniques. The student management group consisted of a group of students from the class with a leader, a recorder, and two members. The group met on a weekly basis to discuss the course, and reported to the instructor on a biweekly basis. The instructor's immediate and thoughtful response to group concerns made the group a partner in real time course evaluation and improvement. The focus group consisted of the entire class. The class met with the Center for Educational Excellence, an independent teaching and learning center at the college, and responded to a set of questions developed by the course instructor. The goal of the focus group was to provide the instructor with meaningful, anonymous, and refereed course evaluation.

The success of the project was another course evaluation technique. The most obvious measure of success of the project was whether the finished project met the requirements of the client. Using this as a measure one author claimed an 80% success rate over two years and twenty-one projects (Goold, 2003). Another measure of success was whether the project was used by the client. Buckley, Kershner, Schindler, Alphonce and Braswell (2004) reported on the successful project that created an augmented communication device for a severely disabled adult, the device was so successful that the next capstone course project was an augmented indication device for a severely disabled child. Another author reported on a successful project that created a paperless form system for local hospital, which would save the hospital over $100,000 in the first year (Bruhn and Camp, 2004).

Daniels et al. (1999) compared several distributed group capstone courses to a conventional capstone course. The authors developed a set of course metrics including: syllabus coverage, student time, staff time, cost, student skill development (language, communication, collaboration, social skills, cultural sensitivity, peer learning) and student motivation. The metrics were applied to a conventional capstone course and several distributed group capstone courses. Student motivation in distributed group capstone courses was higher than student motivation in conventional capstone courses when distributed groups had good social and communication skills. However, student and staff time increased in distributed group capstone courses.

An academic version of the capability maturity model (CMM) was a rigorous course evaluation technique (Collofello, Kantipudi, & Kanko, 1994). By answering a lengthy questionnaire (described in the paper) the instructor could determine a course's level of software engineering maturity:

- Level 1: course did not embrace software engineering as a discipline
- Level 2: course had a software process
- Level 3: course was instructor independent
- Level 4: instructor had quantitative measurements and analysis for the course
- Level 5: instructor continually improved and optimized the software process.

Clearly there were many ways to evaluate a capstone course. We suspect that reflection appeared frequently in the literature because it was straightforward to perform. The problem with reflection was that it was subjective, so the validity and broad applicability of this approach was questionable. Surveys offered a quantitative course evaluation but had a different set of problems. Questions had to be carefully constructed for clarity (e.g. Was the instructors contribution good?) And bias (e.g. How would you evaluate the work you are responsible for?). The survey sample size must also be large enough for statistical validity. The academic CMM offered a rigorous course evaluation and it would be interesting to see the results of a large-scale survey of capstone course instructors using this technique. It would also be interesting to see a student version of the academic CMM that could be used the beginning and end of the course to evaluate course outcomes. Many colleges have a teaching and learning center that offers course evaluation expertise, yet we found only one paper that used a teaching and learning center. We hope to see more literature that uses this valuable resource in the future.

In our capstone course we used several course evaluation techniques: informal feedback from students and clients, quality of course and project artifacts, project success, course evaluations, and the academic CMM. One type of feedback we looked for in particular was unsolicited student feedback after graduation:

> Well you can say "I told you so" in reference to me having to write documents at work. Its all I have been doing for the last month or so! They actually assigned me to go out and look for some new applications and I have to write up what we call a "Whitepaper" on new functionality that the tool(s) provide and functionality we'd have to change in our own system and stuff. Its pretty cool. Those writing classes really helped! You can use me as a testamate to the pain one may have to endure during that class :o)

With course evaluations, we looked at the Likert question response summaries but we also carefully studied written responses:

I felt like too much time was spent building the system and not enough time was spent learning about software engineering. How was the project in this course different from other course projects?

We applied the academic CMM to our own course and found that our course was somewhere between level two and level three. We were unsure that the course was independent of the instructor because of the instructor's experience with large industry projects, previous capstone projects, and teaching. We wondered how effective project mentoring would be with a less experienced instructor.

9. Project: software process models

Many software process models presented in the literature including: waterfall, iterative, agile, personal software process, and team software process.

Waterfall was a common software process model presented in the literature (Beasley, 2003; Bruhn & Camp, 2004; Collofello, 1985; Hribar, 2005; Monroe & Yu, 1998). One of the reasons why waterfall was attractive to course instructors was that each process phase of the project could be synchronized with course lecture. This synchronization was much more challenging with iterative software process models (Chamillard and Braun, 2002). Bruhn and Camp (2004) detailed the use of the waterfall software process model in an industry sponsored two semester capstone course modeled as a consulting company. Students tracked billable hours to show progress at each process phase. Monroe and Yu (1998) described a one semester software engineering course using the waterfall software process model, Ada, and CASE tools. Students learned Ada from scratch and were exposed to new language features at each process phase. For example, at the software design phase students learned about function oriented design, bottom-up design, and object-oriented design through Ada subprogram structures. Hribar (2005) described a homegrown software process model for successful single student projects called SUREFIRE that was based on the waterfall process model. A student choose and developed a unique project, did project planning, did project design by breaking the project into modules and describing each module, developed a schedule, began implementation with skeletal framework to get the system running early, did frequent integration and testing, and reported progress to the instructor through frequent status reports and presentations. Collofello (1985) described a student evaluation technique based on the evaluation of project artifacts at each process phase.

Agile (Bareiss & Griss, 2008; Burge, 2007; Catanio, 2006; Hislop, Lutz, Naveda, 2002; Koster, 2006; Ludi et al., 2005; van Vliet, 2005) and variants like Extreme Programming (XP) (Ali, 2006; Coppit &

Haddox-Schatz, 2005; Dubinsky & Hazzan, 2003; Keefe & Dick, 2004; LeJeune, 2006; Linder, Abbott, & Fromberger, 2006) and Scrum (Alshare, Sanders, Burris, & Sigman, 2007) were also common software process models presented in the literature. Ludi et al. (2005) described the redesign of a course that went from using the waterfall model to the agile model. The iterative nature of the agile model required students to complete an entire development cycle within weeks of starting the course. To help the students quickly understand each phase of the development cycle, the instructor covered the development cycle in two passes, one pass early in the course, and a second longer pass later in the course. Koster (2006) reported on another course that went from using the waterfall model to the agile model. The instructor found better group cohesion because of pair programming, better adoption of coding standards because the students saw benefits across several iterations of the system, and better system stability because testing was performed early and often.

Keefe and Dick (2004) provided a tutorial on the XP software process model, surveyed literature on capstone courses using the XP model, and evaluated the use of the XP model in the author's own course. The tutorial detailed the four core values and twelve core practices of the XP model. The survey found that it was difficult to convince students of the benefits of certain XP model practices. For example, many students resisted the practice of continuously writing and testing the system, although at the end of the project many of them recognized the value of this practice. Students reverted to a non-XP model when under pressure, in particular, pair programming, continuous integration, and upfront testing were abandoned. The author felt that more coaching about the benefits of XP model would help students stick with the process model.

Dubinsky and Hazzan (2003) evaluated the use of the XP software process model across four capstone course sections with different instructors. Instructors participated in an XP model training program before starting the course, were mentored in XP model practices during the course, and reflected on their experiences using XP model after the course. The evaluation found that as the project unfolded, instructors became project leaders and became heavily and personally involved in the project; this involvement allowed the instructors to better evaluate students in the course. The evaluation also found that the XP model values and practices improved communication between group members as compared to the waterfall model. Finally, students did not understand the XP model with just a single semester course, and instructors thought that more instructor training would help with this problem.

Pilskalns (2009) encapsulated the agile software process model within an entrepreneurial process model. Each student group formed an informal startup company complete with a business plan, management group, fund-raising plan, and a plan to create a formal company after the course. The authors reported on the success of one project, a mapping system, that garnered venture capital funding and a patent.

Iterative (Burge, 2007; Chamillard & Braun, 2002; Hutchens & Katz, 1996) and variants like the Rational Unified Process (Way, 2005; Roggio, 2006) were also common software process models presented in the literature. A challenge when using the iterative model was determining how many iterations were possible in a single semester course. Some instructors advocated three iterations (Hutchens & Katz, 1996; Roggio, 2006), while others were skeptical that more than one iteration was possible (Sebern, 1997). To maximize the number of iterations, each iteration should be deadline driven. If a deadline was in jeopardy, features were cut from the iteration. To maximize the effectiveness of an iteration, the instructor provided immediate and detailed feedback on the previous iteration. The first iteration was simple, with a small set of features that would allow focus on group building, tools, and processes necessary to complete the iteration (Hutchens and Katz, 1996).

Roggio (2006) provided a tutorial on the Rational Unified Process (RUP) software process model and evaluated the use of the RUP model in a capstone course. The tutorial detailed the RUP model and linked the model to course project deliverables. The project deliverables were domain independent so the tutorial could be used by other course instructors who were considering the RUP model. Evaluation of the use of the RUP model in the instructors own course found that students had positive feelings about the course. In particular, the students appreciated the emphasis on process rather than product. Students also appreciated being able to use a highly marketable, industrial-strength process and tools on a major course project. On the negative side, the focus on process versus product meant that students did not have very much time to complete the programming portion of the project, and that testing was inadequate.

The Personal Software Process (PSP) model (Borstler et al., 2002; Dick et al., 2000; Gary et al., 2005) and the Team Software Process (TSP) model (Conn, 2004) were less common in the literature. Borstler et al. (2002) reported on experiences using the Personal Software Process (PSP) model at five universities. PSP-lite, a streamlined version of the PSP model, was used in introductory courses, while the full model was used in upper level software engineering courses. The PSP model required significant data collection and analysis, and students complained about this overhead. Trying to explain the benefits of this overhead to a student who had never worked on a large group project did not work. Instead, the authors suggested helping a student to see the benefits of the PSP model through steady improvements in software estimation and quality revealed in data collected and aggregated across the class. The instructor emphasized the activities of collecting and analyzing data rather than the actual data values to avoid data fabrication motivated by grades. Postema, Dick, Miller and Cuce (2000) addressed the overhead for data collection and analysis with automation tools.

Conn (2004) described a version of the Team Software Process model modified for small groups (4–5 students) in an academic setting. The description detailed six phases, including two full development cycles, defined group roles, and listed recurring weekly activities. Evaluation of the use of the TSP model found that students were shocked by the non-programming overhead necessary to build a large group project, that at least two iterations of the development cycle were necessary for students to understand the TSP model, that analysis of the collected data showed a wide variability in student abilities, and that strict deadlines combined with having to work on a group was stressful for some students.

Hadfield and Jensen (2007) discussed a four semester capstone course sequence in which students used many different software process models beginning with PSP in the first semester, and ending with RUP in the last semester. Other software process models that have been presented in the literature included spiral (Bernhart et al., 2006; Concepcion, 1998; Hadfield & Jensen, 2007; Hamilton & Ruocco, 1998; Owen, 1989) and object oriented (Clifton, 1991a; Cordes and Parrish, 1993).

With so many process models to choose from, what was the best model for a capstone course? In a single semester course heavyweight models like waterfall and RUP focused on process at the expense of product, while lightweight processes like agile focused on product at the expense of process. The best software process model was unclear. Perhaps it was not possible to use an industrial strength process model in a single semester capstone course. Instead instructors should consider adapting a process model to the course (Conn, 2004; Hribar, 2005).

In our capstone course we used the waterfall model with some modifications. In the first half of the semester, we concentrated on requirements and design to establish conceptual integrity. Students often expressed concern that they were not allowed to start coding until the middle of the semester. In the second half of the semester, we concentrated on implementation and testing. Implementation was an iterative process with weekly builds that steadily added and enhanced features. At the end of the semester the students realized the importance of establishing conceptual integrity before implementation, but many felt the implementation phase was to compressed and suggested expanding the capstone into a two semester course with the first semester focused on process in the second semester focused on product.

10. Project: phases

Regardless of the software process model, the common software process phases in a capstone course were: requirements, design, implementation, testing, presentation, and maintenance.

Many authors listed the activities common to the requirements phase: gathering requirements, documenting requirements, and revising requirements. In projects where the instructor was the client, it was important to keep the requirements deliberately vague so that the students experienced working on an ill-defined problem (Swift & Neebel, 2003). Several authors noted that students had difficulty documenting requirements, even when models and examples were provided (Chamillard & Braun, 2002; Swift & Neebel, 2003).

Design activities such as UML (Alrifai, 2008; Bergin et al., 1998; Lynch, Flango, Smith, & Lang 2004a; Roggio, 2006), structured design (Bolz & Jones, 1983; Chiang, 2004; Hribar, 2005; Perkins & Beck, 1980), design patterns (Bergin, 2002; Cordes & Parrish, 1993; Rao, 2006), user interface design (Catanio, 2006; Mynatt & Leventhal, 1987; Rao, 2006; Roggio, 2006; van Vliet, 2005), database design (Beasley, 2003; Brazier et al., 2007; Carpenter, Dingle, & Joslin, 2004; Catanio, 2006; Woodside, 2008), and refactoring (LeJeune, 2006; Linder et al., 2006) were mentioned in the literature, but little supporting detail was provided on how these activities could be adapted to a short timeframe of the capstone course. Shumba (2005) discussed UML in a capstone course, but only in the context of comparing student experiences with UML tools. Bergin (2002) found that students needed to use a design pattern in order to understand the pattern. The authors had some success teaching several design patterns with a project involving music.

Teaching user interface design was important because over one half of the typical implementation effort was devoted to the user interface. van Vliet (2005) presented strategies for teaching user interface design with four topics: universal access and internationalization, early user interface design, heuristics and usability studies, and paper prototypes. Before designing the user interface for the course project, students designed, evaluated, and prototyped the user interface on a tutorial project.

Database design was also important because of the prevalence of databases in industry (Carpenter et al., 2004). Woodside (2008) described a project that focused on the design of the medical records system, with particular focus on database design issues like: object-relational schema, UML, procedural SQL, and object traversal using UML. Some instructors also described projects using a three-tier web, application, database architecture that was found in many modern Internet systems (Brazier et al., 2007; Catanio, 2006).

Controversy surrounded the agile process of refactoring. LeJeune (2006) found that refactoring did not work well in the short timeframe of a capstone course with multiple design iterations because each iteration was too fragile and required a complete redesign for the next iteration. Students would have benefited from a significant design of the entire system before starting the implementation. The authors supported their case using citations with similar findings. In contrast, Linder et al. (2006) found that refactoring taught good design when used on small illustrative homework assignments, and the authors supported this finding with different set of citations.

Testing activities like documenting test cases and executing test cases were frequently mentioned in the literature, but with little supporting detail (Adams, 1993; Banios, 1992; Burge, 2007; Jones, 2002; Pournaghshband, 1990; Towhidnejad, 2002). In fact, little was written about how to teach testing to undergraduates in general. If testing was covered at all in a capstone course, it was often at the end of the course during the last-minute scramble to get the project completed. This gave students the impression that testing was an afterthought (Jones, 2002). There were some exceptions in the literature; however, Towhidnejad (2002) reported on a capstone course combined with a graduate testing course. One student from the testing course was assigned to each capstone course group. In the first build, the tester documented a test plan and created test cases while the capstone group wrote project code. In the second build, the tester reviewed code for quality and executed test cases, while the capstone group completed the project code. The instructors found that the first build went smoothly, but the second build had problems. The capstone group felt that the tester got in the way of completing the project, while the tester felt that the capstone group checked in code too late which made it difficult to review the code. The instructor felt that it was tough to do a good job of testing and coding with a single semester course. Schedule coordination between courses was also a problem.

In another course, each capstone group documented test plans, created unit tests for each release, and executed different group test cases (Burge, 2007). Clark (2004) called this approach “peer testing” and reported that students recognized the importance of testing, improved project quality, and worked steadily (instead of procrastinated) on project artifacts.

Most capstone courses required a final presentation at the end of the course, with some courses requiring interim presentations. Liu (2006) constructed a taxonomy of presentations which considered audience (client, faculty, alumni, potential employers, friends, family (Kishline et al., 1998)), style and length (formal – 20 minutes, quick – 5 minutes, poster – variable) and demonstration method (live, screen snapshot). The author also examined three hypotheses about presentations:

- short, frequent, presentations helped students make progress on long-term projects: Student survey results supported this.

- students learned from other student presentations. Student survey results supported this.
- conference style poster presentations were more effective in improving student learning than formal PowerPoint presentations. Student survey results were inconclusive.

Some instructors viewed the final presentation as a form of oral defense (Beasley, 2003; Brazier et al., 2007), while others viewed it as a celebration of student accomplishment (Bruhn & Camp, 2004; Vaughn Jr, 2001). Beasley (2003) described the outline of an oral defense with a panel of judges consisting of the client, another instructor, a campus IT professional, and the course instructor. The student presented a flow of the project through each software process phase, demonstrated the project, and answered questions. The student was also responsible for reserving the room and equipment necessary for the presentation.

Meinke (1987) outlined of two types of capstone presentations. The first presentation occurred early in the semester when the students introduced their products to the class. The following outline was used: project overview, project purpose, project design, project benefits, and questions from the audience. Students prepared a frank evaluation of each presentation, but the presentations were not graded. The presenter was expected to use the feedback to prepare for the second, final presentation. The following outline was used for the final presentation: overview of major project features, advantages/disadvantages of design, problems and how they were overcome, and questions from the audience. Students were discouraged from discussing the entire project and discouraged from presenting code because of time limitations. The instructors found that generally the final presentation was an improvement over the first presentation.

Like the other phases, maintenance was frequently mentioned in the literature with little supporting detail (Adams, 1993; Agrawal & Harriger, 1985; Calhoun, 1987; Chiang, 2004; Tan & Phillips, 2005). The literature that did discuss maintenance was concerned with how to teach maintenance and how to maintain a project after the capstone course. Collofello (1989) outlined four maintenance lectures. The first lecture gave the students an overview of the high cost of maintenance, the three types of maintenance (corrective, enhancement, optimization), and maintenance myths. The second lecture gave the students practical approaches to understanding and documenting and existing software system. The third lecture helped students understand how software changes introduced errors, how to debug software in an organized fashion, and how to document maintenance changes. The final lecture taught students how to validate maintenance changes.

Some instructors used iterative development strategies to teach maintenance. For example, a two semester capstone course had students implement a version of the product in the first semester, then enhance the project in the second semester. Students liked being able to revisit and improve the first version of the project (Brazier et al., 2007).

Boscoa (1991) reversed the project phase order and started with the maintenance phase. Students commented the code, documented the code design, and documented the design requirements. Models were provided for each task. The author felt this experience helped students understand the importance of non-coding project phases. Once the reverse phase process was completed students work on a new project using the traditional phase order process.

Burge (2007) reported on a two semester course that used agile methods in which students completed four iterations of the project. On the third iteration the groups swapped projects. The swap went smoothly because all groups worked on the same basic project. Students were motivated to produce a high quality project because they knew that the other group would be inheriting their project. Challenges while maintaining the other group's project included: uncommented code, difficult installation, inconsistent variable names, unnecessary files, unused code, unidentified files and code inconsistent file naming.

The second maintenance concern addressed by the literature was how to maintain a project after a capstone course had ended. In one course, the instructor checked in with the clients two weeks after the installation of the project. If the project was unsatisfactory then a new group was assigned to the project for the next semester. Some students also worked or volunteered for their clients after the capstone course (Leidig et al., 2006). In another course, students left a maintenance task list with the client at the end of the semester. A rising senior contacted the client, negotiated the maintenance tasks to be completed, and signed a maintenance contract (Beasley, 2003). Concepcion (1998) found problems enhancing the same project semester after semester. Over time, more effort was expended fixing project bugs than enhancing the project. The author believed that this was primarily due to using a procedural approach to design rather than an object oriented approach to design.

Our capstone course students experienced each project phase. In the requirements phase the students produced multiple revisions of the requirements document from a document model, an exemplar document, and extensive comments from the instructor. Although the initial revisions of the document were poor, the students were able to produce an acceptable document with repeated feedback from the instructor.

In the design phase the students worked on the user interface design first. This allowed the students to solidify the project's conceptual integrity for the group, client, and instructor. Design patterns were studied and the instructor worked with students to identify potential design patterns in the project. Students were exposed to UML and created diagrams but these diagrams were usually a combination of UML and custom notation devised by the students. Like the requirements document, the design document required multiple revisions.

Refactoring was discussed along with automated testing, but we found that students were not convinced of the cost-benefit advantages of these techniques. We believe this was due to lack of development experience. One situation where refactoring was helpful was when students ran into implementation difficulties because of overly complex code. We guided the students through refactoring while minimizing the impact on the rest of the project.

Students were exposed to testing through readings and discussion of *The Art of Testing* (Meyers, 1979). Testing was handled differently depending on the size of the group. If the group was small then the instructor was the tester. Otherwise a test group created a software test document, tested each build, and recorded bugs found using a tool like Bugzilla.

We thought maintenance was important but we found the topic impractical to fit into a single semester course. One semester, however, the students were asked to enhance a project from the previous year. The instructor and clients were pleased with the enhanced project, but the students were displeased. The course evaluations for that year were the lowest in most categories in the seven year history of the course. We suspect that the students failed to take ownership of the previous year's project, and the lack of ownership led to lack of motivation. If the client intended to use the project then we connected the capstone students and the client with a rising junior who could maintain the project after the course.

The final presentation was a celebration of the student's hard work. The students invited family and friends, the instructor invited colleagues and administrators, and the client invited colleagues. The students were comfortable and confident during the final presentation because they had presented frequently to the client during the semester.

11. Project: type

We divided projects into five main types: industry, community service, game, simulated, and research.

An industry project provided a solution to a business organization problem with a real client that intended to use the project when it was completed. For example, a password generation system could enhance a business organization information technology infrastructure (Christensen and Rundus, 2003). Industry projects increase the likelihood that the business organization will provide support outside of course projects for the instructor, department, and institution (Gorka, Miller, & Howe, 2007). The literature frequently mentioned the advantages and disadvantages of this project type (see Tables 3 and 4).

Some example industry projects included: satellite data archiving (Hadfield and Jensen, 2007), file delivery protocol (Hadfield and Jensen, 2007), runway damage investigation (Hadfield and Jensen, 2007), information security visualization (Hadfield and Jensen, 2007), unmanned aerial vehicle (Hadfield and Jensen, 2007), distance learning system (Alzamil, 2005), security network upgrade (Gorka et al., 2007), human resources intranet (Tan and Phillips, 2005), heirloom crochet craft

Table 3. Industry project advantages.

Student advantages
– Motivated student early in the course (Buckley et al., 2004)
– Motivated student because the project was going to be used by a real client (Alzamil, 2005; Buckley et al., 2004; Christensen & Rundus, 2003; Gorka et al., 2007; Parkers, Holcombe, & Bell, 1999)
– Motivated student because real client cares about the project (Gorka et al., 2007)
– Motivated student to complete the project (Gorka et al., 2007)
– Provided student with real world experience (Buckley et al., 2004; Parkers et al., 1999)
– Enhanced student professional and technical skills (Bruhn & Camp, 2004; Parberry et al., 2005; Parkers et al., 1999; Tan & Phillips, 2005)
– Exposed student to potential employers (Reichlmay, 2006)
– Exposed student to realistic, vague, and evolving requirements (Reichlmay, 2006; Parkers et al., 1999)
– Exposed student to realistic installation issues (Reichlmay, 2006)
– Added experience to student's resume (Bruhn & Camp, 2004; Parkers et al., 1999)
– Provided student with internship opportunities after the project (Parrish et al., 1998)
– Provided student with rich design possibilities (Buckley et al., 2004)
– Integrated student's prior knowledge (Buckley et al., 2004)
Instructor/Institution advantages
– Provided curriculum feedback (Bruhn & Camp, 2004; Gorka et al., 2007)
– Provided student opportunities (Gorka et al., 2007)
– Provided access to state of the art technology (Gorka et al., 2007)
– Provided instructor research/internship opportunities (Gorka et al., 2007; Reichlmay, 2006)
– Increased instructors sense of gratification (Bruhn & Camp, 2004)
– Helped to establish future partnership with client (Tan & Phillips, 2005)
Client advantages
– Provided access to talented students (Reichlmay, 2006)
– Provided student opportunities (Gorka et al., 2007)
– Provided low-cost solution to problem (Bruhn & Camp, 2004; Christensen & Rundus, 2003; Gorka et al., 2007)
– Provided low-cost examination of alternative solutions to a problem (Gorka et al., 2007)
– Provided fresh ideas that may be applied to other company problems (Gorka et al., 2007)
– Allowed client to observe potential employees (Bruhn & Camp, 2004; Christensen & Rundus, 2003; Gorka et al., 2007)

website (Tan and Phillips, 2005), paperless forms (Bruhn and Camp, 2004), labor tracking and reporting (Christensen and Rundus, 2003), error log compression (Christensen and Rundus, 2003), order tracking system (Christensen and Rundus, 2003), and password generation system (Christensen and Rundus, 2003). See Hagan, Tucker, and Ceddia (1999) for an extensive list.

A community service project provided a solution to a nonprofit organization (often the college) or an individual. For example, a course assessment system could streamline the evaluation of a computer science departments courses (Poger, Schiaffino, & Ricardo 2005). Advantages of community service projects are listed in Table 5.

Some example community service projects that were found in the literature included: course assessment system (Poger et al., 2005), speech system for disabled (Buckley et al., 2004), light-based system for disabled (Buckley et al., 2004), and a student accountability system (Tan and Phillips, 2005).

Game projects provided students with an opportunity to develop a sophisticated game prototype that demonstrated both software engineering and game programming skills. Parberry et al. (2005) reported on a capstone course where groups developed a game engine and a game that used the engine. The groups were interdisciplinary with both computer science and art majors. A contest was held at the end of the semester with judges from the local gaming industry. Some distinct advantages of game projects are listed in Table 6.

Table 4. Industry project disadvantages.

Disadvantages
– Resulted in low quality work (Alzamil, 2005; Liu, 2005)
– Required instructor time to manage student and the client (Parkers et al., 1999)
– Required instructor to release control of the course/project (Parkers et al., 1999)
– Was too big for one semester (Alzamil, 2005; Liu, 2005)
– Competed with other course material (Alzamil, 2005)
– Resulted in different experiences for different groups when there was more than one type of project per course (Liu, 2005)
– Had the risk that client priorities might change which may have resulted in the client removing resources from the project (Gorka et al., 2007; Parkers et al., 1999)

Table 5. Community service project advantages.

– Motivated students by helping people (Buckley et al., 2004; Tan & Phillips, 2005)
– Integrated academics/community service (Tan & Phillips, 2005)
– Taught students ethics/social responsibility (De Rego et al., 2005; Tan & Phillips, 2005)
– Provided client that was willing to tolerate low quality work (Alzamil, 2005)
– Provided clients willing to tolerate project that runs over many capstone courses (Alzamil, 2005)

Some example game projects included: real-time card game (Gehrke et al., 2002), robot simulator (Gehrke et al., 2002), game engine (Bickerstaff & Moro, 2006; Parberry et al., 2005), role-playing game that taught computer science (Distasio and Way, 2007), three-dimensional pool game (Coppit and Haddox-Schatz, 2005), video game using Microsoft's game studio (Ferguson, Rockhold, & Heck, 2008), game of life (Anderson, 1987).

A simulated project had the instructor as a simulated client that never intended to use the project. For example, Carpenter et al. (2004) reported on a series of increasingly complex projects developed during the course including: a soccer team registration system, a book loan system, and a mission control system. Instructors that used a simulated project often subscribed to the belief that students were not mature enough to work with a real customer (Parberry et al., 2005). Simulated projects provided a number of advantages centered around the instructor's ability to control the project (see Table 7).

Some example simulated products included: parallel computing cluster (Heckendorn, 2002; Pokorny, 2005), virtual-reality (Stansfield, 2005), compiler (Clifton, 1991b), relational database (Merzbacher, 2000), kitchen designer (Gehrke et al., 2002), ocean life simulator (Buckley et al., 2004), medical records system (Woodside, 2008), course registration/scheduling system (Alzamil, 2005).

A research project had the student participate in a research experience. The research project could focus on a problem of interest to the student

Table 6. Game project advantages.

– Increased enrollment in major (Parberry et al., 2005)
– Increased retention in major (Parberry et al., 2005)
– Brought in guest speakers from gaming industry (Parberry et al., 2005)
– Trained students for gaming industry careers (Parberry et al., 2005)

Table 7. Simulated project advantages.

– Allowed instructor to control project milestones (Coppit & Haddox-Schatz, 2005)
– Allowed instructor to select project tools (Coppit & Haddox-Schatz, 2005)
– Allowed instructor to control Project management policies (e.g. frequency of group meetings) (Coppit & Haddox-Schatz, 2005)
– Allowed instructor to control scope of project requirements and to reduce requirements as necessary (Coppit & Haddox-Schatz, 2005)
– Allowed instructor to perform dirty tricks on students without upsetting a real client (Gehrke et al., 2002)
– Allowed instructor to focus course on process not product (Gehrke et al., 2002)
– Allowed instructor to use redundant groups to implement the same design which reduces the risk of project failure (Gehrke et al., 2002)
– Provided consistent experience across groups (Gehrke et al., 2002)

or the instructor. Several research projects along with the advantages of these types of projects were described in Section 3.

Fincher et al. (2001) described an alternative taxonomy of projects across the computer science curriculum. We mapped this taxonomy to our own taxonomy: final year individual project (maps to our simulated project), second year group project (not applicable), master of science project (not applicable), software hut project (maps to any project in our taxonomy), research project (maps to our research project), industrial involvement project (maps to our industry project), client project (maps to our industry, community service, and game projects), process-based project (maps to any project in our taxonomy), and capstone project (maps to any project in our taxonomy). The software hut project divided a project across multiple groups, and exposed the student to code integration and collaboration with other groups. The process-based project focused on a software process methodology rather than on completion of a working software system.

All of our capstone projects were community service projects sponsored by college organizations including: campus police, residence life, student admissions and a nonprofit management center. We were open to industry and game projects but none of these project proposal types passed our vetting process. We deliberately avoided simulated and research projects. Despite some advantages, simulated projects lacked authenticity and we felt that this would translate into less student motivation. For research projects, our department offered a separate research project option called the senior honors thesis.

12. Project: documentation

Computer science instructors were anxious about teaching writing (Kay, 1998). To help instructors we proposed a taxonomy of goals and general advice for writing for computer science with an emphasis on capstone projects. The taxonomy had three main goals: writing for learning, writing for academic communication, and writing for industrial communication. Each goal contained a list of writing tasks with a description and citations. Many of the citations provided models for the writing tasks. Details from the third goal: writing for industrial communication are reproduced with additional citations from our capstone literature survey in Table 8 (Dugan & Polanski, 2006):

In addition to the taxonomy of writing goals, we listed general advice for teaching writing Dugan and Polanski (2006):

- Give assignments a real-world context
- Demonstrate that writing was important in industry

Table 8. Writing for industrial communication.

Group	meeting (Alred et al., 2003)
	– agenda for meeting (Alred et al., 2003)
	– minutes for meeting (Alred et al., 2003)
	presentation (Beasley, 2003; Liu, 2006; Meinke, 1987; Polack-Wahl, 2000)
	email (Alred et al., 2003)
	posting to newsgroups/bulletin boards/listserves (William James Hall Computing Services)
	memo (Alred et al., 2003)
Customer	survey/questionnaire (Salant & Dillan, 1994)
	white paper (Orr, 1999)
	presentation (Beasley, 2003; Liu, 2006; Meinke, 1987; Polack-Wahl, 2000)
	website (Alred et al., 2003)
Project management	weekly status report (Alred et al., 2003; Clifton, 1991a; Gotel et al., 2006; Parrish et al., 1998)
	bug report (Collofello, 1989; Pfleeger, 2001)
	group/individual workbook (Perkins & Beck, 1980)
	core documents:
	– proposal (Almstrum, 2005; Meinke, 1987)
	– project plan (Almstrum, 2005; Meinke, 1987)
	– requirements (Almstrum, 2005; Christensen & Rundus, 2003; Concepcion, 1998; Meinke, 1987; Sharon M., 2001; Tan & Phillips, 2005)
	– design (Almstrum, 2005; Christensen & Rundus, 2003; Concepcion, 1998; Meinke, 1987; Sharon M., 2001; Tan & Phillips, 2005)
	– test plan (Almstrum, 2005; Christensen & Rundus, 2003; Concepcion, 1998; Sharon M., 2001)
	– user manual/online help (Adams, 1993; Bremer, 1999; Meinke, 1987; Polack-Wahl, 1999; Sharon M., 2001)
	executive summary (Alred et al., 2003)
	project final report (Dugan & Polanski, 2006; Meinke, 1987; Sharon, 2001)
Career management	resume (Alred et al., 2003; Swift & Neebel, 2003)
	letter (Alred et al., 2003)
	– acceptance letter (Alred et al., 2003)
	– resignation letter (Alred et al., 2003)
	– interview followup letter (Alred et al., 2003)
	job description (Alred et al., 2003)
	employee appraisal (Orr, 1999)
	group mission statement (Orr, 1999)

- Show parallels between the writing process in the software development process
- Require revision
- Conduct peer reviews of assignments
- Use minimal marking
- Use clear, concise, and detailed models for writing tasks
- Orient students to basic rhetorical needs
- Teach types of definitions
- Teach structure and grammar as needed

- Use the face to face conference
- Recommend the Writing Center.

Some instructors believed that students lacked the maturity to be successful at writing tasks, such as requirements and design documentation, despite having given giving students writing task models and examples (Brazier et al., 2007; Chamillard & Braun, 2002; Swift & Neebel, 2003). In contrast, we found that students were successful if the instructor used a clear, concise, and detailed writing task model and followed general advice for teaching writing (Dugan and Polanski, 2006).

Some instructors required students to complete a final report at the end of the project. In addition to being a "writing for industrial communication task", the final report was a also a "reflective writing for learning task." There were many types of reflective writing for learning tasks including: journal, weblog, project report, project presentation, self-evaluation, peer evaluation, and essay (Clear et al., 2001; Dugan & Polanski, 2006; Fincher et al., 2001). Clear et al. (2001) provided a list of questions to reflect upon at the end of the project:

- What initial processes were implemented?
- What conflict resolution took place?
- What were the critical incidents that occurred during the project?
- What was the student's least/most comfortable group role?
- What was the student's most rewarding experience?
- What was the student's most valuable contribution?
- How were the student's contributions crucial to success?

A final report did not have to be the only place where reflective writing occurred during the course. At the beginning of the course the student could reflect on past project experiences. In the middle of the course the student could reflect on current project successes and challenges (Fincher et al., 2001).

Hagan et al. (1999) derived an alternative, orthogonal taxonomy for project documentation: slow-moving (business case, requirements), fast-moving (design, screen design), progress (risk list, task list, time log), legal (client agreement), and final (source code, test, user manual, correspondence).

13. Project: tools

We have organized tools into three broad categories in Table 9.

A major concern was the degree to which the tools supported capstone course goals. Polack-Wahl (2000) and Beck (2005) listed several software engineering education goals (e.g. development environments, software

Table 9. Project tools presented in the literature.

Category	Tool
Requirements/design tools	Rational Rose (Gehrke et al., 2002; Roggio, 2006; Sebern, 1997)
	UML tools (Lynch et al., 2004a; Shumba, 2005)
	generic CASE tools (Alred et al., 2003; Kay, 1998)
Project management tools	course management (Su et al., 2007)
	communication: bulletin board, chat, wiki, social networking (Charltona, Devlina, & Drummond, 2009; Su et al., 2007)
	task management/bug tracking (Charltona et al., 2009; Lancor, 2008; Su et al., 2007)
	source control (Beck, 2005; Gehrke et al., 2002; Su et al., 2007)
	build environment (Su et al., 2007)
Implementation tools	– language: Java (Bergin et al., 1998), Ada (Agrawal & Harriger, 1985; Monroe & Yu, 1998; Owen, 1989)
	.Net (Bickerstaff & Moro, 2006; Woodside, 2008)
	XML (Woodside, 2008)
	Web services (Alrifai, 2008)

process infrastructure, change control), but cautioned against too much exposure to a particular vendor. Other goals supported by tools included: reinforcing a design methodology, supporting project management, supporting implementation, improving writing and oral skills, and experiencing a realistic, resume–worthy technology. Tools detracted from course goals by consuming limited resources: money for tool purchase, hardware, and upgrades; instructor time for tool installation, configuration, and administration; student time for tool training (Kay, 1998; Roggio, 2006; Su et al., 2007). Lancor (2008) reported a significant reduction in money and instructor time needed with a project management tool that used software as a service technology.

We were very careful with tools in our capstone course because we did not want the tools to interfere with the course goals. However, we have found several useful tools: a wiki for project communication and documentation, a version control system for project artifacts, and an issue tracking system for bugs and enhancements.

14. Project: groups

In a nationwide survey of engineering capstone courses, over 80% of the responding departments taught group skills (Todd et al., 1995). This survey reinforced our literature survey finding that group skills were a common goal of capstone courses. Group skills were important because most students would be working on group projects after graduation (Bareiss & Griss, 2008; Knight & Horton, 2005; Lancor, 2008; Poger et al., 2005; Todd et al., 1995).

Teaching group skills was challenging because students and instructors lacked group experiences; many computer science students were introverts, and instructors had not been taught how to teach group skills (Cushing et al., 2003). In addition most undergraduate courses valued individual work over group work (Fincher et al., 2001; Waite, Jackson, Diwan, & Leonardi, 2004). A survey of software engineering students found that when students were placed on a group as the only method for teaching group skills, a number of problems arose including: lack of communication, poor leadership, failure to compromise, procrastination, flawed integration testing, lack of cooperation, lack of confidence, scheduling conflicts, and personal problems (Pournaghshband, 1990; Saiedian, 1996).

An instructor could address the challenge of teaching group skills by deeply understanding the skills, demonstrating the skills in the classroom, using group exercises to reinforce the skills, and organizing group aspects of the course project. Cushing et al. (2003) provided a deeper understanding of group skills by organizing them into three categories: tactical (organizing work, making decisions, solving problems), inter/intra-personal (resolving conflicts, communicating with others, using interpersonal skills, giving and getting feedback, and participating in meetings and presentations), and leadership. By acting as a collaborator and facilitator rather than a lecturer, the instructor demonstrated group skills, Waite et al. (2004) improved group skills, and improved course artifact grades. Group exercises such as the egg drop problem; the observer, communicator, and builder problem; the Belbin test; Myers-Briggs test; and software reviews reinforced group skills Cushing et al. (2003); Fincher et al. (2001); Waite et al. (2004). To organize group aspects of the course project, Cushing et al. (2003) ranked the importance of different group skills against software process phases to help the instructor focus on teaching the right skills at the right time. To avoid these problems the instructor helped the students to: understand the project requirements; understand how communication, cooperation and compromise could achieve a common goal; and cooperate in a carefully selected and balanced group (Pournaghshband, 1990).

The rest of this section will discuss group issues including: group selection, group size and group organization.

14.1. Groups: selection

Three group selection techniques were presented in the literature: instructor, student, and random. The goal of these techniques was to create balanced, well functioning project groups according to the criteria in Table 10.

The instructor group selection technique used the instructor to select students for each group. Our literature survey found that this technique was the most common, in contrast to Dutson et al. (1997) (a broader

Table 10. Group selection criteria.

Category	Criteria
Academic/ technical	– technical skills (e.g. programming languages, tools) (Ecker et al., 2004; Perkins & Beck, 1980; Pournaghshband, 1990; Scott & Cross, 1995; Sharon M., 2001) – project specific skills (e.g. domain expertise, familiarity with client) (Ecker et al., 2004; Richards, 2009; Scott & Cross, 1995) – academic skills (e.g. GPA) (Ecker et al., 2004; Richards, 2009; Scott & Cross, 1995) – instructor's experience with student (Richards, 2009) – major/minor (Richards, 2009) – motivation (Richards, 2009) – programming ability (Richards, 2009)
Group	– personality/group chemistry (friends/enemies, meek/overbearing) (Bruhn & Camp, 2004; Hribar, 2005; Pournaghshband, 1990; Richards, 2009; Scott & Cross, 1995) – time management (timely/procrastinating) (Hribar, 2005) – group skills (Ecker et al., 2004; Richards, 2009; Sharon M., 2001) – leadership skills (Clifton, 1991a; Ecker et al., 2004) – group role preference (Clifton, 1991a; Hribar, 2005) – psychological profile (Myers-Briggs Personality Type Indicator, Perry Levels of Ethical and Intellectual Development, Belbin test, Kolb Learning Style Index, Hermann Brain Dominance Instrument) (Belbin, 2010; Dutson et al., 1997; Richards, 2009; Scott & Cross, 1995) – gender/race/nationality/culture (Emanuel & Worthington, 1989; Richards, 2009)
Other	– schedule (courses, job, clubs, sports, full-time/part-time student) (Hribar, 2005; Richards, 2009; Scott & Cross, 1995; Sharon, 2001) – access to transportation (Emanuel & Worthington, 1989; Richards, 2009) – project preference (Hribar, 2005; Perkins & Beck, 1980; Scott & Cross, 1995)

survey of engineering capstone courses) which found the student group selection technique more common. Instructors used their own subjective evaluation, subjective evaluation from other faculty, class surveys, student resumes, student transcripts (although Fincher et al. (2001) cautioned that this was not a good indicator of how an individual will perform in a group), and student interviews to collect group criteria. The advantages of this technique were that it created the most balanced groups using the above criteria, that it put students together who might not ordinarily work with each other, and that it most accurately reflected how groups were selected in the real world. The disadvantages of this technique were the significant amount of effort required by the instructor and student complaints about how the group was constructed (Scott and Cross, 1995).

The student group selection technique allowed students to choose their own groups. Familiarity within a group had some advantages: the project started sooner, friction within the group was reduced, and pressure existed not to let group members down. In addition, little effort was required by the instructor, and students would not complain about how the groups were constructed. The disadvantages of this technique were that a group composed of friends did not always perform well because the group lacked balance in other criteria, that unfamiliar members were excluded from discussions and decisions, that workload amongst friend members was not evenly distributed and that this selection technique was not used to select groups in the real world (Saiedian, 1996; Scott & Cross, 1995).

The random student group selection technique randomly assigned students to each group. The advantages of this technique included: that little effort was required by the instructor, and that it put students together who might not ordinarily work with each other. There were many disadvantages to using this technique, however, because the technique did not take into account any criteria for a balanced group. For example, the members of one group might have consisted of students who had poor technical skills, who were enemies, and who had incompatible schedules (Scott & Cross, 1995; Vaughn Jr, 2001).

Richards (2009) surveyed group selection in computer science project courses as well as other disciplines. The author's selection criteria was included in Table 10. The author also discussed heterogeneous versus homogenous groups and the ethics of combining high achieving and low achieving students into the same group (high achieving students often earned a lower grade in this combination).

Student selected groups were often homogenous, but Richards (2009) cited a study by Rutherfoord (2001) that showed heterogeneous groups were more effective. To select a heterogeneous group Richards (2009) cited work by Rutherfoord (2001) using the Myers-Briggs test and the Keating temperament instrument and by Nelson, Bass, and Vance (1994) using the managing group formation process. Ultimately Richards (2009) chose to develop a simple group selection survey and selection algorithm. For each group the author tried to have: two competent programmers, introverts and extroverts, and a balance of English proficiency. The author also found some criteria had to be homogenous for group harmony: schedules, expected grade, and expected hours of effort.

There was little evidence to support the benefits of any of the group selection techniques. One exception was a study by Emanuel and Worthington (1989) of a Civil Engineering capstone course (summarized in Dutson et al. (1997)) that examined students in groups formed using different combinations of academic skills and project preference. The students in groups with heterogeneous academic skills and homogenous project preferences had the best attitudes towards the course, but it was not clear that this resulted in a more successful project.

14.2. Groups: size

There was no agreement in the literature on an ideal group size because both small and large groups had advantages and disadvantages (see Table 11).

There was no agreement in the literature on the quantity of students in a small group versus a large group. Clearly a group size of one was a small group, and Christensen and Rundus (2003) found that a single member group had the advantage of increased student motivation if the student was able to select the project. A disadvantage was the lack of faculty recognition when the single member group project was treated as an independent study/senior thesis outside of the regular classroom.

Moving up in size, we found groups of three to ten students to be the most common in the literature with disagreement on whether these were small or large groups (Alrifai, 2008; Barnes, Richter, Powell, Chaffin, & Godwin 2007; Catanio, 2006; Collofello & Woodfield, 1982; Leidig et al., 2006; Sun & Decker, 2004; Tan & Phillips, 2005). Our finding was in line with Dutson et al. (1997) (a broader survey of engineering capstone courses) which found that 38% had one to three students, 49% had four to six students, and 7% had groups with seven or more students.

Table 11. Group size advantages and disadvantages.

Group size	Advantage/disadvantage
Small	– required less instructor effort to manage (Coppit & Haddox-Schatz, 2005; Tan & Phillips, 2005) – reduced likelihood of non-contributing student (Sharon M., 2001) – allowed each student to be involved in more parts of the project (Sharon M., 2001) – allowed ad-hoc project management (Collofello & Woodfield, 1982) – lacked realism (Northrop, 1989) – allowed only toy projects (Christensen & Rundus, 2003) – required instructor to devise many projects for course (Emanuel & Worthington, 1989)
Large	– required project management (group leaders, agendas, frequent meetings, subgroups, milestones) (Christensen & Rundus, 2003; Collofello & Woodfield, 1982; Gillies & Gillies, 1999) – required software engineering (up front design, documentation, version control) (Christensen & Rundus, 2003; Coppit & Haddox-Schatz, 2005) – required communication (Coppit & Haddox-Schatz, 2005; Gillies & Gillies, 1999) – allowed group to experience real world large group issues (Christensen & Rundus, 2003; Clifton, 1991a; Coppit & Haddox-Schatz, 2005; Gillies & Gillies, 1999; Northrop, 1989) – allowed group to work on large, real world project (Christensen & Rundus, 2003) – required more instructor effort to manage (Coppit & Haddox-Schatz, 2005; Tan & Phillips, 2005) – increased likelihood of non-contributing student (Emanuel & Worthington, 1989; Sharon, 2001)

Christensen and Rundus (2003), for example, reported many advantages when using groups of four to six students. By the end of the course, students believed that project management and software engineering skills were as important as technical skills, that the course enhanced their ability to work as part of a group, and that they had gained valuable industry preparation.

Moving up further up in size we found large groups consisting of the entire class (Bolz & Jones, 1983; Coppit & Haddox-Schatz, 2005; Northrop, 1989). Coppit and Haddox-Schatz (2005), for example, reported on a group of thirty students and found that mandatory group meetings were critical to the success of the project, despite the difficulties of conflicting student schedules. The instructor received reports from group managers, which reduced the need for interaction with each group member. Students ranked (in order) version control, weekly meetings, intermediate deadlines, issue tracking, and discussion forums as the most important aspects of the course. Bolz and Jones (1983), for another example, reported on a group of twenty-three students and found that it was critical to assign a clear role to each student on the group, and to devise a map which detailed the assignments for each student for each milestone. The point value for an assignment varied depending on the student's role in the assignment. Student evaluations of the course were very positive despite having worked an average of over 100 hours on the project. Students also appreciated being able to talk about the large group project in employment interviews, and employers appreciated the large group project experience the students received.

The most extreme example of a large group consisted of over 100 students from many disciplines including computer science, electrical engineering, civil engineering, business, marketing, and economics. The goal of the project was to build an autonomous vehicle for laying down lines on a playing field. The authors reported many problems including overworked students, lack of discipline diversity on subgroups, inconsistent experience across groups, feature creep, and conflicting course and project goals across departments. The authors concluded that this large group approach was not a good course model (Neumann and Woodfill, 1998).

14.3. Groups: organization

Groups were found to be more effective when each student was assigned an explicit and compatible group role (Fincher et al., 2001). Over twenty years ago Tomayko (1987) provided a list of group roles for an undergraduate software engineering group project. A summary of this list is presented with an abbreviated definition, synonyms, and additional literature references.

- *Principal Architect* (Technical Lead, Systems Analyst): Responsible for the creation of the software product (Beasley et al., 2004; Conn, 2004; Northrop, 1989; Tan & Phillips, 2005).
- *Project Administrator* (Leader, Group Leader, Project Leader, Manager, Project Manager, Requirements Reengineer, Planning Manager, Manager, Client Liaison, and Accountant): Responsible for resource allocation and tracking (Conn, 2004; Davis, 2001; Ecker et al., 2004; Gehrke et al., 2002; Mynatt & Leventhal, 1987; Northrop, 1989; Perkins & Beck, 1980; Tan & Phillips, 2005; Žagar et al., 2008).
- *Configuration Manager* (System Administrator, Revision Controller, Inventory Controller): Responsible for change control (Beasley et al., 2004; Northrop, 1989).
- *Quality Assurance Manager* (Tester): Responsible for the overall quality of the released product (Conn, 2004; Gehrke et al., 2002; Northrop, 1989; Tan & Phillips, 2005).
- *Test and Evaluation Engineer* (Tester): Responsible for testing and evaluating individual modules (Gehrke et al., 2002; Northrop, 1989; Tan & Phillips, 2005).
- *Verification and Validation Engineer* (Tester): Responsible for creating and executing test plans to verify and validate the software (Gehrke et al., 2002; Northrop, 1989; Tan & Phillips, 2005).
- *Designer* (Systems Designer): Responsible for developing aspects of the design as specified by the Principal Architect (Gehrke et al., 2002; Northrop, 1989; Tan & Phillips, 2005).
- *Implementor* (Group Member, Coder, Reengineer): Responsible for implementing the individual modules of the design (Conn, 2004; Gehrke et al., 2002; Northrop, 1989).
- *Documentation Specialist*: Responsible for the appearance and clarity of all documentation and for the creation of user manuals (Gehrke et al., 2002; Northrop, 1989).
- *Maintenance Engineer* (Installer): Responsible for creating a guide to the maintenance of the delivered product (Gehrke et al., 2002; Northrop, 1989).

Others (in addition to Tomayko (1987)'s list):

- *Support Manager* (Customer Support Representative): Responsible for customer support (Conn, 2004; Polack-Wahl, 1999).
- *Database Administrator*: Responsible for design and implementation of database component of project (Tan and Phillips, 2005).

These group roles were reasonably comprehensive and mirrored the more modern group roles such those found in the SEI Team Software

Process (Humphrey, 1999; Towhidnejad, 2002). For a large class, multiple students could be assigned to the same role, while for a smaller class multiple roles could be assigned to the same student. Students were assigned to roles using instructor selection techniques (see Groups: Selection) (Chamillard & Braun, 2002; Clifton, 1991a; Ecker et al., 2004; Gehrke et al., 2002; Northrop, 1989; Perkins & Beck, 1980; Sharon M., 2001) and student selection techniques (see Groups: Selection) (Bruhn and Camp, 2004). Fincher et al. (2001) recommended that students take the Belbin survey which uncovered strength in eight groupwork areas: implementer, coordinator, shaper, plan and resource investigator, monitor-evaluator, group player, and finisher. Gehrke et al. (2002) recommended that in addition to an assigned role, each student should be responsible for specific system features which the student followed from requirements through implementation and testing.

The most commonly discussed role in the literature was the project administrator. In addition to students from the course, this role could be filled by course alumni (Beasley et al., 2004), teaching assistants (Beasley et al., 2004; Žagar et al., 2008), or the instructor (Bareiss & Griss, 2008; Hain & Jodis, 1992; Towhidnejad, 2002). In one capstone course sequence, seniors were project administrators for groups of juniors, and juniors were implementers of design specifications created by seniors (Gary et al., 2005). Ecker et al. (2004) believed that students were reluctant to become the project administrator because of the extra work; however, students could be incentivized to take on the extra work with extra credit (Clifton, 1991a). Typical responsibilities of a project administrator included: weekly meetings with the group, reviewing the activities in progress during the past week, reviewing the activities for the next week, comparing what the group said they would do and what group actually did, reviewing time cards submitted by group members, isolating problems and creating action plan to solve the problems, and updating the instructor on group progress (Perkins and Beck, 1980).

With the globalization of software development distributed groups brought additional realism to the group experience. Bolz and Jones (1983), for example, reported on groups consisting of fifteen students per group, one group per section, and three to four course sections working on the same project. Each group was responsible for certain features, but groups were forced to interact with each other to integrate the features. Students learned about groupwork, interpersonal relations, and project management. Way (2005), reported on a similar course with five groups distributed across two course sections. To simulate distributed groups, one section was located in virtual California, and the other group located in virtual New Hampshire. Groups in different sections used electronic communication and were not allowed to meet face-to-face without approval from the instructor. Course alumni felt that the course prepared them for large group projects, and industry colleagues were favorably impressed with this distributed group approach.

Some instructors collaborated with instructors at other universities to create truly distributed groups. The Runestone project was a multiyear study of distributed Swedish and American capstone course groups totaling hundreds of students. During the study the course was improved to address unequal instructor access and issues with the simulated project (Last, Daniels, Hause, & Woodroffe 2002). Common problems with distributed groups included: cultural differences, time zone differences, incompatible schedules, conflicting instructor goals, and interpersonal relations (Adams et al., 2003; Fincher et al., 2001; Gotel, Scharff, & Seng 2006; Last et al., 2002; Žagar et al., 2008).

Communication and coordination were critical in distributed groups. Hause et al. (2003) found that high performing groups communicated less and made less explicit decisions than low performing groups. The authors speculated that the quality of the communication was superior in high-performing groups. Gotel et al. (2006) found that the most effective form of communication between groups was mediated communication where a single member on each of the two groups acted as an intermediary between groups.

The most effective distributed groups were those in which students got to know each other at a personal level and humor could be an effective technique for making that personal connection (Gotel et al., 2006; Last et al., 2002).

Interdisciplinary groups brought realism to the group experience and then learn from each other (Catanio, 2006; Ecker et al., 2004; Hadfield & Jensen, 2007; Polack-Wahl, 1999; Swift & Neebel, 2003). Hadfield and Jensen (2007) reported that computer science majors perceived other majors in the capstone course as less capable, but as time progressed the computer science majors began to recognize and appreciate the similarities between disciplines. A survey at the end of one course comprised of CS and IT majors found that IT majors had a better appreciation of programming (Catanio, 2006).

Problems arose when an interdisciplinary course lacked balance among disciplines. Buchheit et al. (1999), for example, found that computer science majors thought the course was too easy and did most of the project work, while other majors thought that course was too difficult and that there was too much emphasis on implementation. One way to provide balance was to define roles for each discipline. Parrish et al. (1998) reported on a two semester interdisciplinary course with CS, CE, and MIS majors. In the first semester, MIS majors assessed business requirements; CS majors designed an architecture, built components, and seeded repositories; and CE majors choose hardware, and helped CS majors. In the second semester, MIS majors conducted acceptance testing and CS/CE majors integrated components and built a running system. In another course, the instructor assigned CS majors the role of developer, and IT majors the role of customer (Gotel et al., 2006). Another way to provide balance was to have all majors take the same classes together, but complete discipline specific projects (Kishline et al., 1998). Yet another way to provide balance was to make sure that course content, homework, and projects were relevant to each discipline (Ecker et al., 2004; Hadfield & Jensen, 2007).

If multiple instructors were involved in the interdisciplinary course, then it was important that they communicated on a continuous basis about course expectations and student evaluations to avoid conflicts between instructors (Ecker et al., 2004; Fincher et al., 2001; Neumann & Woodfill, 1998).

Another group organization issue was the mapping of groups to projects. A single group/single project map worked well for small groups, but was more challenging in larger groups due to the potential for communication breakdown. A multiple group/single project mapping worked well if the project could be subdivided into independent components. This mapping also encouraged cooperation. A multiple group/single project mapping reduced demand on the instructor and created healthy competition between groups. Finally a multiple group/multiple project mapping increased demand on the instructor and required that project complexity be fairly balanced (Saiedian, 1996).

14.4. Groups: reflection

In our capstone course the entire class formed a single project group. Our course enrollment was generally small (less than 10 students) so our students took on multiple group roles. Generally the students selected their own roles with more demanding roles (e.g. designer, implementor) selected by stronger students. Students with demanding roles often complained of unfair workload and after several semesters with these complaints we assigned a "contract implementor" role to students with less demanding roles. Implementers were allowed to give contract implementers small programming tasks.

There were several years when we had larger (10 or more students) course enrollment. We used the instructor selection technique to assign students to groups based on our subjective evaluation of ability and personality type. In one course we had two groups compete against each other and we asked client choose the best final product. At the end of that semester both groups told us that the product would have been better if the groups had worked cooperatively rather than competitively. In courses with large enrollment that followed, we used a combination of competition and cooperation. Groups competed in the requirements and design phase with the client using the best design. Then the groups were merged for the implementation and testing phase. At the end of the design phase the client generally leaned towards one design but also liked parts of the competing design. The designs were merged before implementation.

We did not experiment with distributed groups, because our course was already demanding for both the instructor and students. But we recognized that global distributed groups will be something that most of our graduates will experience. It would be interesting to see work that described how to manage the additional workload when using distributed groups. Like distributed groups, our graduates will work on interdisciplinary groups but we were again concerned about the additional workload and would like see work that describes how to manage this additional workload.

15. Project: instructor administration

Instructor administration was concerned with how an instructor solicited projects, vetted projects, dealt with group/client conflicts, and dealt with legal issues. We distinguished instructor administration from project management, which was concerned with how the group planned, organized, and executed the project.

15.1. Soliciting projects

Instructors considered a variety of potential clients when soliciting projects: instructor (Clear et al., 2001), University (Clear et al., 2001), nonprofit organizations (Clear et al., 2001; Leidig et al., 2006), business organizations (Clear et al., 2001; Reichlmay, 2006), government organizations (Hadfield and Jensen, 2007), students (Beasley, 2003), friends (Beasley, 2003), family (Beasley, 2003), alumni, and design competitions (Dutson et al., 1997).

Strategies for solicitation included: advertising (newspaper (Clear et al., 2001), web (Reichlmay, 2006), e-mail (Beasley, 2003)); contacting organizations via a department maintained list (Clear et al., 2001); and handing out information sheets at professional and academic workshops (Clear et al., 2001; Reichlmay, 2006). The solicitation was made months prior to the course to give potential clients time to respond and to give the instructor time to vet the projects (Beasley, 2003). The solicitation could include a description of the course, a description of past projects, and should ask potential clients to provide a brief description of the proposed project.

15.2. *Vetting projects*

In order to vet projects, instructors developed project descriptions; established criteria for projects, clients, and groups; and selected projects.

A single paragraph from a potential client was sufficient for a project description, however, some instructors developed the descriptions further (Leidig et al., 2006; Reichlmay, 2006). Leidig et al. (2006), for example, described a series of meetings with potential clients to understand how the client conducted business, how the client used information technology, and what projects the client was considering. The instructor and potential client then drafted a requirements document (sometimes very detailed) for one or more projects.

The instructor should establish criteria for a good project and client (see Table 12).

The instructor should also establish good criteria for assigning students to projects. One instructor had groups rank potential projects.

Table 12. Good project and client criteria.

Project challenge
– Had novel, nontrivial: requirements, design, algorithms, application programming interfaces, user interface components, database components (Beasley, 2003; Christensen & Rundus, 2003; Dutson et al., 1997; Hadfield & Jensen, 2007; Leidig et al., 2006).
– Avoided trivial projects (Beasley, 2003).
– Had sufficient size for class wide participation (Clifton, 1991a).
– Had a reasonable chance of completion (Dutson et al., 1997).
– Had reasonable workload (Hadfield & Jensen, 2007).
Project pedagogy
– Provided excellent learning experience (Leidig et al., 2006).
– Motivated students to do the best work (Leidig et al., 2006).
– Provided fair/group assessment (Leidig et al., 2006).
– Fostered interpersonal communication skills (Hadfield & Jensen, 2007).
– Emphasized application of theory (Dutson et al., 1997).
– Encouraged reading of literature (Dutson et al., 1997).
Project management
– Was not critical to organization (Reichlmay, 2006).
– Minimized unnecessary outside influences (Leidig et al., 2006).
– Provided ongoing sources for new projects (Clifton, 1991a; Leidig et al., 2006).
– Did not involve proprietary information (Christensen & Rundus, 2003; Dutson et al., 1997).
Project client
– Accepted cost of software and hardware to support project (Clear et al., 2001).
– Accepted possibility project may have poor quality or may even be unusable (Clear et al., 2001).
– Had access to funds necessary to improve project (Clear et al., 2001).
– Was willing to do a guest lecture (Christensen & Rundus, 2003).
– Allowed students to do a client site presentation (Christensen & Rundus, 2003).
– Provided a corporate mentor who meets frequently with group (Bruhn & Camp, 2004).

In the case of a tie, each group submitted a memo detailing why the group was the best choice for the project (Reichlmay, 2006). Another instructor allowed students to choose from a list of potential projects, but pushed a stronger/weaker student towards a more challenging than/less challenging project to increase the chances of a rewarding experience (Reichlmay, 2006). Clements (2007) took a contrary approach which matched a student to a project that challenged a student's weaker skills and knowledge.

To select one or more projects for the course, the instructor vetted each potential project according to project, client, and group assignment criteria. The vetting may be done by a single instructor, by all department instructors teaching the capstone courses, or by all faculty in the department (Bruhn & Camp, 2004; Reichlmay, 2006). If a project did not meet the criteria then the instructor could work with a potential client to resubmit the project proposal at a later date (Reichlmay, 2006).

15.3. Dealing with group/client conflict

Clear et al. (2001) discussed how an instructor dealt with group/client conflict in detail. One instructor goal should be to avoid conflicts in the first place. Strategies for avoiding conflict included: insuring the project, client, and group meet criteria established for the project; gathering frequent feedback from the group and client (especially when a conflict occurred;) mediating/facilitating immediately when conflict occurred; and establishing a procedure for resolving conflict. If conflicts became severe, the instructor should have had a policy for how the client or group could back out of the project.

15.4. Dealing with legal issues

Instructors dealt with a number of legal issues including: liability, confidentiality, finances, and ownership. In the United States, the Volunteer Protection Act protected volunteers (i.e. students, instructors) from project liability, but did not protect the volunteers organization (i.e. the university) (Sun and Decker, 2004). Leidig et al. (2006) offered a sample liability agreement between a university and the client.

There were differing opinions on how a client confidentiality agreement should be handled. Some instructors felt that if confidentiality agreement was required, then the project should not be considered (Christensen & Rundus, 2003; Dutson et al., 1997). Others felt that some degree of client confidentiality was reasonable, provided that the agreement did not interfere with the goals of the course. A litmus test of confidentiality was to ask the client if some of the details of the project could be published or presented (Clear et al., 2001). Leidig et al. (2006)

offered a sample confidentiality agreement between students and the client.

There were many possibilities for project ownership at the end of the course: student, instructor, university, client, open source community, or any combination of these. In some countries, copyright laws automatically assigned ownership of the project to the creator. If the student was the owner, then a sponsor may need permission to use a third-party to maintain and enhance the software (Clear et al., 2001). Leidig et al. (2006) offered a sample ownership agreement that gave the client ownership of the project.

Clear et al. (2001) provided a detailed discussion of finance matters the instructor should consider. The institution should be responsible for the basic course costs, but other arrangements were possible for project specific costs. The client could pay for all project costs, pay a fixed fee, donate equipment, donate money to the department, or donate money to a scholarship. Student compensation should also be addressed. Some institutions allowed students to be compensated for academic coursework, while other institutions did not. Compensation made liability more of an issue because the students may be viewed as consultants or employees rather than volunteers protected by the Volunteer Protection Act.

15.5. Reflection

In the fall, we solicited projects from students and organizations for our spring capstone course. If the project was initiated by students, we required the sponsoring client. Clients were vetted using several criteria:

- the client was enthusiastic about the project
- the client was committed to face-to-face meetings for gathering requirements, finalizing requirements, finalizing design, reviewing beta software, and attending a final presentation
- the client was available for questions and clarifications during the semester
- the client was committed to hiring a rising junior for the summer to maintain and enhance the project
- the client was committed to using the project if the project met requirements
- the project was not critical to the operation of the client's organization.

These criteria grew from our experiences with the capstone course over time.

We did not have serious group/client conflict in our projects. We believe this was because we used Clear et al. (2001) 's strategy of feedback gathering to avoid conflict.

We had a number of liability, confidentiality, and ownership issues. If we were unsure about an issue, we consulted the College's legal counsel. In one instance involving student records, the capstone students were asked to sign a confidentiality agreement similar to Leidig et al. (2006). Regarding finances, because all of our projects were community service projects, we shared the development cost with the client (e.g. software licensing, hardware, internet service, etc.) for up to a year.

16. Conclusion

In this survey we examined literature related to undergraduate computer capstone courses and we organized the literature around course and project issues. For course issues we examined course models, learning theories, course goals, course topics, student evaluation, and course evaluation.

A common capstone course model was a single semester course with software engineering lectures and a large group project. Variants of this model stretched the course from two to eight semesters which allowed the instructor to cover more software engineering topics, focus on process, and focus on product. Students in our single semester capstone course have suggested stretching the course to two semesters because they felt that process and product components of the course were too compressed. Alternate course models included a research experience or special topics course, but these models have not mainstreamed because they lack the authentic experience needed by industry-bound students.

The traditional capstone course used lectures to teach software engineering theory and a large group project to apply that theory. From a learning theory perspective, this course could be modeled using the transferring model and weak problem-based learning theories. Literature that described the capstone course based on formal learning theory was sparse, and evidence to support the efficacy of a particular learning theory was weak. Of the four major learning theories (behaviorism, cognitivism, constructivism, humanism) constructivism was discussed almost exclusively in the literature. More research is needed on learning theories and capstone courses.

Many capstone course goals and topics competed for the instructors attention. We suggested a prioritization scheme based on citation frequency in the capstone course literature. Using this scheme we found that our own capstone course needed additional attention on testing and maintenance. While this prioritization scheme may have some merit, the larger question remains: How can a capstone course cover the wide variety of goals and topics described in the literature?

A single group project grade was unfair because of the wide variability of student capstone project performance. Many student evaluation techniques were proposed based on evaluators (instructor, student, student peers, group leader, client, department), artifacts (version control management logs, meeting minutes, and student journals), and a weighted individual/group grade. Regardless of the evaluation technique used, students needed continuous evaluation throughout course to prevent small problems from becoming large ones.

There were many ways to evaluate a capstone course: reflection, survey, academic CMM, focus group, teaching and learning center. Reflections (subjective) and surveys (poorly constructed questions, small sample size) had well-known disadvantages. The rigorous academic CMM held promise, but has yet to be broadly applied across capstone courses. We suggested creating a student version of the academic CMM for pre/post course evaluation. Many colleges have a teaching and learning center that offers course evaluation expertise, yet we found only one paper that used this valuable resource.

In addition to course issues, we examined project issues including: software process models, software process phases, project type, documentation, tools, groups, and instructor administration.

The waterfall, agile, and iterative software process models were common in the capstone course literature. The waterfall model allowed the instructor to synchronize lectures with each process phase during the semester, but the heavyweight model's artifacts and product were challenging to complete in a single semester. The agile and iterative models allowed students to see the benefits of good software engineering across multiple iterations of the project, but the models challenged the instructor to provide enough detail about process phases for early iterations of the project, and challenged the instructor to fit multiple iterations into a single semester course. Perhaps it is not possible to use industrial-strength software process models in a single semester capstone course, but it might be possible to create formal adaptations of these models for the short single semester time-frame.

Regardless of the software process model, the software process phases frequently presented in the literature were: requirements, design, implication, testing, presentation, and maintenance. Although these process phases were touched upon in the literature, there was little supporting detail. A few notable exceptions examined user interface design, testing, presentation, and maintenance in detail. Additional research needs to be done to show how to execute each software phase in the compressed timeframe of a single semester capstone course.

Capstone course projects could be divided into several types: industrial, community service, simulated, and research. For each project type we provided a definition, listed advantages and disadvantages, and listed representative projects. In our own capstone course, we avoided simulated and research projects. Despite some advantages, simulated projects lacked authenticity and we felt that this would translate into less student motivation. Additionally, most of the simulated project advantages related to instructor control over which centered the learning around the instructor rather than the student. Constructivism advocates student-centered learning. For research projects, our department offered a separate research project option called the senior honors thesis.

Instructors were anxious about teaching writing. To help instructors, we included a taxonomy of goals and general advice for writing for computer science with an emphasis on capstone courses (Dugan and Polanski, 2006). Instructors should provide writing models and exemplars to help students understand expectations for project artifacts like the requirements document. Timely feedback and multiple revisions of project artifacts should help to improve student writing skills. Finally to prepare for the writing-intensive capstone course, instructors should ensure that students experience writing for computer science in earlier computer science courses.

We organized capstone course tools presented in the literature into three broad categories: requirements and design, project management, and implementation. A major concern was the degree to which tools supported course goals. Tools could expose students to realistic resume-worthy technology, but could also consume limited resources like money and instructor/student time. In our capstone course we have found several useful tools: a wiki for project communication and documentation, a version control system for project artifacts, and an issue tracking system for bugs and enhancements.

Teaching group skills to computer science students in a capstone course was challenging because computer science students were often introverts and instructors had not been taught how to teach group skills. To increase the likelihood of a successful project group the instructor should consider group selection, size, and organization. Group selection involved defining criteria for a balanced group and deciding who will choose the group members. Interestingly, there was little objective evidence to support the benefits of a specific group selection technique. The literature discussed group sizes anywhere from one to one hundred students with the most common size ranging between three and ten students. We grouped group sizes into small and large and listed advantages and disadvantages for each size. Group roles defined twenty years ago by Tomayko (1987) were still useful for capstone group organization. The role of project administrator was frequently discussed in detail in the literature. Some instructors advocated the use of distributed and interdisciplinary groups to enhance product realism, but it was unclear how the instructor and students managed the additional workload in an already demanding course.

In addition to project management, instructors performed several administrative tasks including: soliciting projects, vetting projects, dealing with group/client conflict, and dealing with legal issues. Projects should be solicited a semester in advance of the course to give the instructor time to vet the project. Group/client conflict could be avoided by keeping the lines of communication open between the group, client, and instructor. The instructor could minimize conflict by acting immediately when conflict was reported by the group or client. Legal issues could include: liability, confidentiality, finances, and ownership. In our own capstone course, we often encountered legal issues so it was important to have a good relationship with legal counsel.

We conclude with some final thoughts. The literature we reviewed ($\sim$200 papers) was several times smaller than all of the literature available on this topic and it is possible that we missed some important work. The survey reflected the views of published instructors, not the entire population of instructors. For views that reflected a sample population of instructors see Fincher et al. (2001). There are many opportunities to contribute research to the growing body of computer science capstone literature including: aligning formal learning theory with the capstone course, prioritizing capstone course goals and topics, rigorously evaluating the capstone course, and conducting a large scale survey capstone instructors regarding course and project issues.

References

AACU. (1990). *The challenge of connecting learning. Liberal learning and the arts and sciences major*, volume 1. Washington, DC: Association of American Colleges and Universities.

ABET. (2005). *Criteria for accrediting engineering programs*. Baltimore, MD: Accreditation Board for Engineering and Technology.

Abran, A., Moore, J.E.E., Bourque, P., & D.R. (Eds.). (2004). *Guide to the software engineering body of knowledge – 2004 version*. Los Alamitos, CA: IEEE Computer Society.

Adams, E.J. A project-intensive software design course. (1993). In *SIGCSE '93: Proceedings of the 24th SIGCSE technical symposium on computer science education* (pp. 112–116). New York, NY: ACM.

Adams, L., Daniels, M., Goold, A., Hazzan, O., Lynch, K., & Newman, I. (2003). Challenges in teaching capstone courses. In *ITiCSE 03: Proceedings of the 8th annual conference on innovation and technology in computer science education* (pp. 219–220). New York, NY: ACM.

Agrawal, J.C., & Harriger, A.R. (1985). Undergraduate courses needed in ADA and software engineering. In *SIGCSE '85: Proceedings of the 16th SIGCSE technical symposium on computer science education* (pp. 266–281). New York, NY: ACM.

Akingbehin, K., Maxima, B., & Tsuia, L. (1994). A capstone design course based on computing curricula 1991. *Computer Science Education*, *5*, 229–240.

Ali, M.R. (2006). Imparting effective software engineering education. *SIGCSE Software Engineering Notes*, *31*, 1–3.

Almstrum, V. Cs373: S2s project documentation standards. Retrieved from: http://www.cs.utexas.edu/users/almstrum/cs373/fa05/doc-stds

Alred, F., Brusaw, C., & Oliu, W. (2003). *Handbook of technical writing* (7th ed.). Boston, MA: St. Martin's Press.

Alrifai, R. (2008). A project approach for teaching software architecture and web services in a software engineering course. *Journal of Computing Sciences in Colleges*, *23*, 237–240.

Alshare, K.A., Sanders, D., Burris, E., & Sigman, S. (2007). How do we manage student projects? Panel discussion. *Journal of Computing Sciences in Colleges*, *22*, 29–31.

Alzamil, Z. (2005). Towards an effective software engineering course project. In *ICSE '05: Proceedings of the 27th international conference on software engineering* (pp. 631–632). New York, NY: ACM.

Anderson, E. (1987). Survival: A tale of a senior project. *SIGCSE Bulletin*, *19*, 22–24.

Bagert, D. Forum for advancing software engineering education. (1998). Retrieved from http://www.cs.ttu.edu/fase/v8n09.txt

Banios, E. (1992). An engineering practices course. *IEEE Transactions on Education*, *35*, 286–293.

Bareiss, R., & Griss, M. (2008). A story-centered, learn-by-doing approach to software engineering education. In *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on computer science education* (pp. 221–225). New York, NY: ACM.

Barnes, T., Richter, H., Powell, E., Chaffin, A., & Godwin, A. (2007). Game2learn: Building cs1 learning games for retention. In *ITiCSE '07: Proceedings of the 12th Annual SIGCSE conference on innovation and technology in computer science education* (pp. 121–125). New York, NY: ACM.

Beasley, R.E., Cupp, J.W., Sanders, D., & Walker, E. (2004). Developing senior capstone projects: Panel discussion. *Journal of Computing Sciences in Colleges*, *20*, 26–26.

Beasley, R.E. (2003). Conducting a successful senior capstone course in computing. *Journal of Computing Sciences in Colleges*, *19*, 122–131.

Beck, J. (2005). Using the CVS version management system in a software engineering course. *Journal of Computing Sciences in Colleges*, *20*, 57–65.

Belbin, R.M. (2010). *Team roles at work* (2nd ed.). Burlington, MA: Butterworth Heinemann.

Bergin, J. (2002). Teaching on the wiki web. In *ITiCSE '02: Proceedings of the 7th annual conference on innovation and technology in computer science education* (pp. 195–195). New York, NY: ACM.

Bergin, J., Naps, T.L., Bland, C.G., Hartley, S.J., Holliday, M.A., Lawhead, P.B., ... Teräsvirta, T. (1998). Java resources for computer science instruction. In *ITiCSE-WGR '98: Working group reports of the 3rd annual SIGCSE/SIGCUE ITiCSE conference on integrating technology into computer science education* (pp. 14–34). New York, NY: ACM.

Bernhart, M., Grechenig, T., Hetzl, J., & Zuser, W. (2006). Dimensions of software engineering course design. In *ICSE '06: Proceedings of the 28th international conference on software engineering* (pp. 667–672). New York, NY: ACM.

Bickerstaff, D.D., & Moro, R. (2006). A light-weight software engineering approach to developing small-scale information systems using .net 2.0. *Journal of Computing Sciences in Colleges*, *22*, 27–29.

Bolz, R.E., & Jones, L.G. (1983). A realistic, two-course sequence in large scale software engineering. In *SIGCSE '83: Proceedings of the 14th SIGCSE technical symposium on computer science education* (pp. 21–24). New York, NY: ACM.

Borstler, J., Carrington, D., Hislop, G., Lisack, S., Olson, K., & Williams, L. (1991). Teaching PSP: Challenges and lessons learned. *IEEE Software*, *19*, 42–48.

Boscoa, M.F. (1991). Teaching software engineering by reverse engineering. *Computer Science Education*, *2*, 117–130.

Brazier, P., Garcia, A., & Vaca, A. (2007). A software engineering senior design project inherited from a partially implemented software engineering class project. In *FIE '07: Proceedings of the 37th annual frontiers in education conference* (pp. F4D–7–F4D–12). New York, NY: ASEE).

Bremer, M. (1999). *The user manual: How to research, write, test, edit and produce a software manual* (1st ed.). Grass Valley, CA: UnTechnical Press.

Bridgeman, N. (2008). Capstone projects: An NACCQ retrospective. In *NACCQ '08: Proceedings of the 21st annual conference of the national advisory committee on computing qualification* (pp. 12–16). NACCQ Research and Support Working Group: Hamilton, New Zealand.

Bruhn, R.E., & Camp, J. (2004). Capstone course creates useful business products and corporate-ready students. *SIGCSE Bulletin*, *36*, 87–92.

Buchheit, N., Ruocco, A., & Welch, D. (1999). A pilot senior CS capstone sequence for CS majors and nonmajors. In *FIE '99: Proceedings of the 29th annual frontiers in education conference*, volume 3 (pp. 13B3/13B3/12–13B3/15B3/15). New York, NY: ASEE).

Buckley, M., Kershner, H., Schindler, K., Alphonce, C., & Braswell, J. (2004). Benefits of using socially-relevant projects in computer science and engineering education. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on computer science education* (pp. 482–486). New York, NY: ACM.

Budgen, D., & Tomayko, J. Norm Gibbs and his contribution to software engineering education through the sei curriculum modules. In *CSEET '03: Proceedings of the 16th conference on software engineering education and training* (pp. 3–13). New York, NY: IEEE.

Burge, J. (2007). Exploiting multiplicity to teach reliability and maintainability in a capstone project. In *CSEET '07: Proceedings of the 20th conference on software engineering education and training* (pp. 29–36). New York, NY: IEEE.

Calhoun, J. (1987). Distribution of software engineering concepts beyond the software engineering course. In *SIGCSE '87: Proceedings of the 18th SIGCSE technical symposium on computer science education* (pp. 233–237). New York, NY: ACM.

Capon, P. (1999). Maximizing learning outcomes of computer science projects. *Computer Science Education*, *9*, 184–199.

Carpenter, T.E., Dingle, A., & Joslin, D. (2004). Ensuring capstone project success for a diverse student body. *Journal of Computing Sciences in Colleges*, *20*, 86–93.

Carver, D.L. (1987). Recommendations for software engineering education. In *SIGCSE '87: Proceedings of the 18th SIGCSE technical symposium on computer science education* (pp. 228–232). New York, NY: ACM.

Catanio, J.T. (2006). An interdisciplinary practical approach to teaching the software development life-cycle. In *SIGITE '06: Proceedings of the 7th conference on information technology education* (pp. 3–8). New York, NY: ACM.

Chamillard, A.T., & Braun, K.A. (2002). The software engineering capstone: Structure and tradeoffs. In *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on computer science education* (pp. 227–231). New York, NY: ACM.

Charltona, T., Devlina, M., & Drummond, S. (2009). Using Facebook to improve communication in undergraduate software development teams. *Computer Science Education*, *19*, 273–292.

Chiang, C.-C. (2004). Teaching a formal method in a software engineering course. In *MSCCC '04: Proceedings of the 2nd annual conference on mid-south college computing* (pp. 39–52). Little Rock, AK: Mid-South College Computing Conference.

Christensen, K., & Rundus, D. (2003). The capstone senior design course: An initiative in partnering with industry. In *FIE '03: Proceedings of the 33rd annual frontiers in education conference*, volume 3 (pp. S2B–12–17). New York, NY: ASEE.

Clark, M.C., & Boyle, R.D. (1999). A personal theory of teaching computing through final year projects. *Computer Science Education*, *9*, 200–214.

Clark, N. (2004). Peer testing in software engineering projects. In *ACE '04: Proceedings of the 6th conference on Australasian computing education* (pp. 41–48). Darlinghurst, Australia: Australian Computer Society, Inc.

Clear, T. (2007). Computing capstone projects and the role of failure in education. *SIGCSE Bulletin*, *39*, 13–15.

Clear, T., Goldweber, M., Young, F.H., Leidig, P.M., & Scott, K. (2001). Resources for instructors of capstone courses in computing. *SIGCSE Bulletin*, *33*, 93–113.

Clements, A. (2007). Work in progress – Matching capstone projects to the student. In *FIE '07: Proceedings of the 37th annual frontiers in education conference* (pp. S1D–7–S1D–8). New York, NY: ASEE.

Clifton, J.M. (1991a). An industry approach to the software engineering course. In *SIGCSE '91: Proceedings of the 22nd SIGCSE technical symposium on computer science education* (pp. 296–299). New York, NY: ACM.

Clifton, J.M. (1991b). An industry approach to the software engineering course. In *SIGCSE '91: Proceedings of the 22nd SIGCSE technical symposium on computer science education* (pp. 296–299). New York, NY: ACM.

Collofello, J.S., Kantipudi, M., & Kanko, M.A. (1994). Assessing the software process maturity of software engineering courses. In *SIGCSE '94: Proceedings of the 25th SIGCSE technical symposium on computer science education* (pp. 16–20). New York, NY: ACM.

Collofello, J.S., & Woodfield, S.N. (1982). A project-unified software engineering course sequence. In *SIGCSE '82: Proceedings of the 13th SIGCSE technical symposium on computer science education* (pp. 13–19). New York, NY: ACM.

Collofello, J.S. (1985). Monitoring and evaluating individual team members in a software engineering course. In *SIGCSE '85: Proceedings of the 16th SIGCSE technical symposium on computer science education* (pp. 6–8). New York, NY: ACM.

Collofello, J.S. (1989). Teaching practical software maintenance skills in a software engineering course. In *SIGCSE '89: Proceedings of the 20th SIGCSE technical symposium on computer science education* (pp. 182–184). New York, NY: ACM.

Concepcion, A.I. (1998). Using an object-oriented software life-cycle model in the software engineering course. In *SIGCSE '98: Proceedings of the 29th SIGCSE technical symposium on computer science education* (pp. 30–34). New York, NY: ACM.

Conn, R. (2004). A reusable, academic-strength, metrics-based software engineering process for capstone courses and projects. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on computer science education* (pp. 492–496). New York, NY: ACM.

Coppit, D., & J.M. Haddox-Schatz. (2005). Large team projects in software engineering courses. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on computer science education* (pp. 137–141). New York, NY: ACM.

Cordes, D., & Parrish, A. (1993). Incorporating re-use into a software engineering course with ADA. In *WADAS '93: Proceedings of the 10th Annual Washington ADA Symposium on ADA* (pp. 109–114). New York, NY: ACM.

Cushing, J., Cunningham, K., & Freeman, G. (2003). Towards best practices in software teamwork. *Journal of Computing Sciences in Colleges*, *19*, 72–81.

Daniels, M., Berglund, A., & Petre, M. (1999). Reflections on international projects in undergraduate CS education. *Computer Science Education*, *9*, 256–267.

Davis, M.C. (2001). A student's perspective of a capstone course. *Journal of Computing Sciences in Colleges*, *16*, 151–167.

Dawson, R. (2000). Twenty dirty tricks to train software engineers. In *ICSE '00: Proceedings of the 22nd international conference on software engineering* (pp. 209–218). New York, NY: ACM.

De, J., Rego, F.R., Zoltowski, C., Jamieson, L., & Oakes, W. (2005). Teaching ethics and the social impact of engineering within a capstone course. In *FIE '05: Proceedings of the 35th annual frontiers in education conference* (pp. S3D–1–5). New York, NY: ASEE.

DeClue, T. (2007). A comprehensive capstone project in computer science I: Getting the (instant) message. *Journal of Computing Sciences in Colleges*, *22*, 56–61.

Dick, M., Postema, M., & Miller, J. (2000). Teaching tools for software engineering education. In *ITiCSE '00: Proceedings of the 5th Annual SIGCSE/SIGCUE ITiCSE conference on innovation and technology in computer science education* (pp. 49–52). New York, NY: ACM.

Distasio, J., & Way, T. (2007). Inclusive computer science education using a ready-made computer game framework. In *ITiCSE '07: Proceedings of the 12th annual SIGCSE conference on innovation and technology in computer science education* (pp. 116–120). New York, NY: ACM.

Dubinsky, Y., & Hazzan, O. (2003). Extreme programming as a framework for student-project coaching in computer science capstone courses. In *SwSTE '03: Proceedings of the IEEE international conference on software: Science, technology and engineering* (pp. 53–59). New York, NY: IEEE.

Dubinsky, Y., & Hazzan, O. (2005). A framework for teaching software development methods. *Computer Science Education*, *15*, 275–296.

Dugan, R.F., Jr., & Polanski, V.G. (2006). Writing for computer science: A taxonomy of writing tasks and general advice. *Journal of Computing Sciences in Colleges*, *21*, 191–203.

Dutson, A.J., Todd, R.H., Magleby, S.P, and Sorensen, C.D. (1997). A review of literature on teaching engineering design through project-oriented capstone courses. *Journal of Engineering Education*, *86*, 17–28.

Ecker, P.S., Caudill, J., Hoctor, D., & Meyer, C. (2004). Implementing an interdisciplinary capstone course for associate degree information technology programs. In *CITC5 '04: Proceedings of the 5th conference on information technology education* (pp. 60–65). New York, NY: ACM.

Emanuel, J., & Worthington, K. (1989). Team oriented capstone design course management: A new approach to team formulation and evaluation. In *FIE '89: Proceedings of 19th annual frontiers in education conference* (pp. 229–234). New York, NY: ASEE.

Ferguson, E., Rockhold, B., & Heck, B. (2008). Video game development using xna game studio and c#.net. *Journal of Computing Sciences in Colleges*, *23*, 186–188.

Fincher, S., Petre, M., & Clark, M. editors. (2001). *Computer science project work: Principles and pragmatics*. London, UK: Springer-Verlag.

Fisher, T.G., & Abunawass, A.M. (1994). Computer ethics: A capstone course. In *Proceedings of the conference on ethics in the computer age* (pp. 74–79). New York, NY: ACM.

Ford, G. (1982). A software engineering approach to first year computer science courses. In *SIGCSE '82: Proceedings of the 13th SIGCSE technical symposium on computer science education* (pp. 8–12). New York, NY: ACM.

Ford, G. (1991). The SEI undergraduate curriculum in software engineering. In *SIGCSE '91: Proceedings of the 22nd SIGCSE technical symposium on computer science education* (pp. 375–385). New York, NY: ACM.

Ford, G. (1994). The progress of undergraduate software engineering education. *SIGCSE Bulletin*, *26*, 51–55.

Fox, D. (1983). Personal theories of teaching. *Studies in Higher Education*, *8*, 151–163.

Freeman, P., Wasserman, A.I, & Fairley, R.E. (1976). Essential elements of software engineering education. In *ICSE '76: Proceedings of the 2nd international conference on software engineering* (pp. 116–122). Los Alamitos, CA: IEEE Computer Society Press.

Gary, K., Gannod, B., Gannod, G., Koehnemann, H., Lindquist, T., & Whitehouse, R. (2005). Work in progress – the software enterprise. In *FIE '05: Proceedings of the 35th annual frontiers in education conference* (pp. S3J–7–8). New York, NY: ASEE.

Gehrke, M., Giese, H., Nickel, U.A., Niere, J., Tichy, M., Wadsack, J.P., & Zundorf, A. (2002). Reporting about industrial strength software engineering courses for undergraduates. In *ICSE '02: Proceedings of the 24th international conference on software engineering* (pp. 395–405). New York, NY: ACM.

Gillies, A., & Gillies, L. (1999). A large-scale software engineering group project at the University of the West of England, Bristol. *Computer Science Education*, *9*, 268–280.

Goold, A. (2003). Providing process for projects in capstone courses. In *ITiCSE '03: Proceedings of the 8th annual conference on innovation and technology in computer science education* (pp. 26–29). New York, NY: ACM.

Gorka, S., Miller, J.R, & Howe, B.J. (2007). Developing realistic capstone projects in conjunction with industry. In *SIGITE '07: Proceedings of the 8th ACM SIGITE conference on information technology education* (pp. 27–32). New York, NY: ACM.

Gotel, O., Scharff, C., & Seng, S. (2006). Preparing computer science students for global software development. In *FIE '06: Proceedings of the 36th annual frontiers in education conference* (pp. 9–14). New York, NY: ASEE.

Grable, R. (2003). Information characteristics for the curriculum. *SIGCSE Bulletin*, *35*, 74–77.

Hadfield, S.M., & Jensen, N.A. (2007). Crafting a software engineering capstone project course. *Journal of Computing Sciences in Colleges*, *23*, 190–197.

Hadjerrouit, S. (2005). Constructivism as guiding philosophy for software engineering education. *SIGCSE Bulletin*, *37*, 45–49.

Hagan, D., Tucker, S., & Ceddia, J. (1999). Industrial experience projects: A balance of process and product. *Computer Science Education*, *9*, 215–229.

Hain, T.F., & Jodis, S.M. (1992). Senior design project course: A case study. *Computer Science Education*, *3*, 169–187.

Hamilton, A., & Ruocco, A. (1998). Implementation issues for CS majors and nonmajors in a senior CS capstone sequence. In *FIE '98: Proceedings of the 28th annual frontiers in education conference*, Volume 3 (pp. 1012–1015). New York, NY: ASEE.

Hauer, A., & Daniels, M. (2008). A learning theory perspective on running open ended group projects (OEGPS). In *ACE '08: Proceedings of the tenth conference on Australasian computing education* (pp. 85–91). Darlinghurst, Australia: Australian Computer Society, Inc.

Hause, M., Petre, M., & Woodroff, M., (2003). Performance in international computer science collaboration between distributed student teams. In *FIE '03: Proceedings of 33rd annual conference on frontiers in education*, Volume 3 (pp. S1F–18). Los Alamitos, CA: IEEE Computer Society.

Heckendorn, R.B. (2002). Building a Beowulf: Leveraging research and department needs for student enrichment via project based learning. *Computer Science Education*, *12*, 255–273.

Henry, S. (1983). A project oriented course on software engineering. In *SIGCSE '83: Proceedings of the 14th SIGCSE technical symposium on computer science education* (pp. 57–61). New York, NY: ACM.

Hislop, G.W., Lutz, M.J., Naveda, J.F., McCracken, W.M., Mead, N.R, & Williams, L.A. (2002). Integrating agile practices into software engineering courses. *Computer Science Education*, *12*, 169–185.

Hoffman, A.A. (1978). A survey of software engineering courses. In *SIGCSE '78: Proceedings of the 9th SIGCSE technical symposium on computer science education* (pp. 80–83). New York, NY: ACM.

Hogan, J.M., & Thomas, R. (2005). Developing the software engineering team. In *ACE '05: Proceedings of the 7th Australasian conference on computing education – Volume 42* (pp. 203–210). Darlinghurst, Australia: Australian Computer Society, Inc.

Hribar, M.R. (2005). Sure fire programming: A general framework for independent projects in computer science. *Journal of Computing Sciences in Colleges*, *21*, 257–266.

Huang, T. (2003). The game of Go: An ideal environment for capstone and undergraduate research projects. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on computer science education* (pp. 84–88). New York, NY: ACM.

Humphrey, W.S. (1999). *Introduction to the team software process* (1st ed.). Boston, MA: Addison-Wesley Professional.

Hutchens, D.H., & Katz, E.E. (1996). Using iterative enhancement in undergraduate software engineering courses. In *SIGCSE '96: Proceedings of the 27th SIGCSE technical symposium on computer science education* (pp. 266–270). New York, NY: ACM.

Jones, E.L. (2002). Testing in the capstone course – reusable patterns for a value-added experience. *Journal of Computing Sciences in Colleges*, *17*, 142–144.

Judith, W.C., Bair, B., Börstler, J., Timothy, L.C, & Surendran, K. (2003). Client sponsored projects in software engineering courses. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on computer science education* (pp. 401–402). New York, NY: ACM.

Kay, D.G. (1998). Computer scientists can teach writing: An upper division course for computer science majors. In *SIGCSE '98: Proceedings of the 29th SIGCSE technical symposium on computer science education* (pp. 117–120). New York, NY: ACM.

Keefe, K., & Dick, M. (2004). Using extreme programming in a capstone project. In *ACE '04: Proceedings of the 6th conference on Australasian computing education* (pp. 151–160). Darlinghurst, Australia: Australian Computer Society, Inc.

Kishline, C., Wang, F., & Aggoune, E.-H. Competency-based engineering design courses development. In *Northcon '98: Proceedings IEEE technical applications conference* (pp. 202–207). New York, NY: IEEE.

Knight, J., & Horton, T. (2005). Evaluating a software engineering project course model based on studio presentations. In *FIE '05: Proceedings of the 35th annual frontiers in education conference* (pp. S2H–21–26). New York, NY: ASEE.

Koster, B. (2006). Agile methods fix software engineering course. *Journal of Computing Sciences in Colleges*, *22*, 131–137.

Lancor, L. (2008). Collaboration tools in a one-semester software engineering course: What worked? What didn't? *Journal of Computing Sciences in Colleges*, *23*, 160–168.

Last, M.Z., Daniels, M., Hause, M.L, & Woodroffe, M.R. (2002). Learning from students: Continuous improvement in international collaboration. In *ITiCSE '02: Proceedings of the 7th annual conference on innovation and technology in computer science education* (pp. 136–140). New York, NY. ACM.

Leidig, P.M., Ferguson, R., & Leidig, J. (2006). The use of community-based non-profit organizations in information systems capstone projects. In *ITICSE '06: Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (pp. 148–152). New York, NY: ACM.

LeJeune, N.F. (2006). Teaching software engineering practices with extreme programming. *Journal of Computing Sciences in Colleges*, *21*, 107–117.

Lejk, M., & Wyvill, M. (1996). A survey of methods of deriving individual grades from group assessments. *Assessment and Evaluation in Higher Education*, *21*, 267–280.

Leonard, D.C. (1979). *Learning theories A to Z.* Westport, CT: Greenwood Press.

Leslie, J., Waguespack, J., & Hass, D.F. (1984). A workbench for project oriented software engineering courses. In *SIGSCE '84: Proceedings of the 15th SIGCSE technical symposium on computer science education* (pp. 137–145). New York, NY: ACM.

Linder, S.P., Abbott, D., & Fromberger, M.J. (2006). An instructional scaffolding approach to teaching software design. *Journal of Computing Sciences in Colleges*, *21*, 238–250.

Liu, C. (2005). Enriching software engineering courses with service-learning projects and the open-source approach. In *ICSE '05: Proceedings of the 27th international conference on software engineering* (pp. 613–614). New York, NY: ACM.

Liu, C. (2006). Software project demonstrations as not only an assessment tool but also a learning tool. In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on computer science education* (pp. 423–427). New York, NY: ACM.

Long, A.N., & Thoreson, S.A. (1986). Incorporating software engineering techniques into a computer architecture course. In *SAC '86: Proceedings of the 1986 workshop on applied computing* (pp. 9–11). New York, NY: ACM.

Ludi, S., Natarajan, S., & Reichlmayr, T. (2005). An introductory software engineering course that facilitates active learning. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on computer science education* (pp. 302–306). New York, NY: ACM.

Lynch, A., Flango, D., Smith, R., & Lang, M. (2004a). Experiences of using rational rose/visio for UML modeling in an undergraduate software engineering course: A student perspective. *Journal of Computing Sciences in Colleges*, *19*, 353–356.

Lynch, K., Goold, A., & Blain, J. (2004b). Students' pedagogical preferences in the delivery of it capstone courses. In *InSITE 2004: Informing science and IT education joint conference* (pp. 431–442). Santa Rosa, CA: Informing Science Institute.

Mazlack, L.J. (1981). Using a sales incentive technique in a first course in software engineering. In *SIGCSE '81: Proceedings of the 12th SIGCSE technical symposium on computer science education* (pp. 37–40). New York, NY: ACM.

Meer, G.L.V, and Sigwart, C.D. (1985). Beyond a first course in software engineering. *SIGCSE Bulletin*, *17*, 26–29.

Meinke, J.G. (1987). Augmenting a software engineering projects course with oral and written communication. In *SIGCSE '87: Proceedings of the 18th SIGCSE technical symposium on computer science education* (pp. 238–243). New York, NY: ACM.

Merzbacher, M. (2000). Teaching database management systems with Java. In *SIGCSE '00: Proceedings of the 31st SIGCSE technical symposium on computer science education* (pp. 31–35). New York, NY: ACM.

Meyers, G.J. (1979). *The art of software testing* (1st ed.). Hoboken, NJ: John Wiley and Sons.

Miller, R.L., & Olds, B.M. (1994). A model curriculum for a capstone course in multidisciplinary engineering design. *Journal of Engineering Education*, *83*, 311–316.

Modesitt, K.L., Bagert, D., & Werth, L. (2001). Academic software engineering: What is and what could be? Results of the first annual survey for international SE programs. In *ICSE '01: Proceedings of the 23rd international conference on software engineering* (pp. 643–652). Washington, DC: IEEE Computer Society.

Monroe, J., & Yu, H. (1998). A software engineering using ADA 95 course. *ADA Letters*, *18*, 86–91.

Mwanza, D. (2001). Where theory meets practice: A case for an activity theory based methodology to guide computer system design. In *INTERACT 2001: Proceedings of the eighth IFIP TC 13 conference on human-computer interaction* (pp. 342–349). Oxford, UK: IOS Press.

Mynatt, B., & Leventhal, L. (1987). Profile of undergraduate software engineering courses: Results from a survey. In *SIGCSE '87: Proceedings of the 18th SIGCSE technical symposium on computer science education* (pp. 523–528). New York, NY: ACM.

Nelson, R.E., Bass, K.C., and Vance, C. (1994). Managed group formation: An approach to team formation in policy courses. *Journal of Education for Business*, *70*, 25–29.

Neumann, W., & Woodfill, M. (1998). A comparison of alternative approaches to the capstone experience: Case studies versus collaborative projects. In *FIE '98: Proceedings of the 28th annual frontiers in education conference*, Volume 1 (pp. 470–474). New York, NY: ASEE.

Northrop, L.M. (1989). Success with the project-intensive model for an undergraduate software engineering course. In *SIGCSE '89: Proceedings of the 20th SIGCSE technical symposium on computer science education* (pp. 151–155). New York, NY: ACM.

Orr, T. (1999). Genre in the field of computer science and computer engineering. *IEEE Transactions on Professional Communication*, *42*, 32–37.

Owen, G.S. (1989). An ADA-based software engineering course. In *SIGCSE '89: Proceedings of the 20th SIGCSE technical symposium on computer science education* (pp. 213–216). New York, NY: ACM.

Parberry, I., Roden, T., & Kazemzadeh, M.B. (2005). Experience with an industry-driven capstone course on game programming: Extended abstract. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on computer science education* (pp. 91–95). New York, NY: ACM.

Parkers, H., Holcombe, M., & Bell, A. (1999). Keeping our customers happy: Myths and management issues in client-led student software projects. *Computer Science Education*, *9*, 230–241.

Parnas, D.L. (1972). A course on software engineering techniques. In *SIGCSE '72: Proceedings of the 2nd SIGCSE technical symposium on computer science education* (pp. 154–159). New York, NY: ACM.

Parrish, A., Borie, R., Cordes, D., Dixon, B., Hale, D., Hale, J., … Sharpe, S. (1998). Computer engineering, computer science and management information systems: Partners in a unified software engineering curriculum. In *CSEET '98: Proceedings of the 11th conference on software engineering education and training* (pp. 67–75). New York, NY: IEEE.

Perkins, T.E., & Beck, L.L. (1980). A project-oriented undergraduate course sequence in software engineering. In *SIGCSE '80: Proceedings of the 11th SIGCSE technical symposium on computer science education* (pp. 32–39). New York, NY: ACM.

Petricig, M., & Freeman, P. (1984). Software engineering education: A survey. *SIGCSE Bulletin*, *16*, 18–22.

Pfleeger, S. (2001). Testing the system: Problem report forms. In *Software engineering: Theory and practice* (2nd ed.). Upper Saddle River, NJ: Prentice-Hall.

Pilskalns, O. (2009). An entrepreneurial approach to project-based courses. *Computer Science Education*, *19*, 193–204.

Poger, S., Schiaffino, R., & Ricardo, C. (2005). A software development project: A student-written assessment system. *Journal of Computing Sciences in Colleges*, *20*, 229–238.

Pokorny, K.L. (2005). Constructing a Beowulf cluster as a pedagogical tool. *Journal of Computing Sciences in Colleges*, *20*, 104–105.

Polack-Wahl, J. (2000). It is time to stand up and communicate [computer science courses]. In *FIE '00: Proceedings of the 30th annual frontiers in education conference*, Volume 1 (pp. F1G/16–F1G/21). New York, NY: ASEE.

Polack-Wahl, J.A. (1999). Incorporating the client's role in a software engineering course. In *SIGCSE '99: The proceedings of the thirtieth SIGCSE technical symposium on computer science education* (pp. 73–77). New York, NY: ACM.

Postema, M., Dick, M., Miller, J., & Cuce, S. (2000). Tool support for teaching the personal software process. *Computer Science Education*, *10*, 179–193.

Pournaghshband, H. (1990). The students' problems in courses with team projects. In *SIGCSE '90: Proceedings of the 21st SIGCSE technical symposium on computer science education* (pp. 44–47). New York, NY: ACM.

Ramakrishan, S. (2003). Muse studio lab and innovative software engineering capstone project experience. In *ITiCSE '03: Proceedings of the 8th annual conference on innovation and technology in computer science education* (pp. 21–25). New York, NY: ACM.

Rao, M.K. (2006). Storytelling and puzzles in a software engineering course. In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on computer science education* (pp. 418–422). New York, NY: ACM.

Reichlmay, T.J. (2006). Collaborating with industry: Strategies for an undergraduate software engineering program. In *SSEE '06: Proceedings of the 2006 international workshop on summit on software engineering education* (pp. 13–16). New York, NY: ACM.

Rhodus, T., & Hoskins, J. (1995). Toward a philosophy for capstone courses in horticulture. *Horticulture Technology*, *5*, 175–178.

Richards, D. (2009). Designing project-based courses with a focus on group formation and assessment. *ACM Transactions on Computing Education*, *9*, 40.

Rising, L. (1989). Removing the emphasis on coding in a course on software engineering. In *SIGCSE '89: Proceedings of the 20th SIGCSE technical symposium on computer science education* (pp. 185–189). New York, NY: ACM.

Roggio, R.F. (2006). A model for the software engineering capstone sequence using the rational unified process [text registered]. In *ACM-SE 44: Proceedings of the 44th Annual Southeast Regional Conference* (pp. 306–311). New York, NY: ACM.

Rutherfoord, R.H. (2001). Using personality inventories to help form teams for software engineering class projects. In *ITiCSE '01: Proceedings of the 6th annual SIGCSE/SIGCUE ITiCSE conference on innovation and technology in computer science education* (pp. 73–76). New York, NY: ACM.

Saiedian, H. (1996). Organizing and managing software engineering team projects. *Computer Science Education*, *7*, 109–132.

Salant, P., & Dillan, D. (1994). *How to conduct your own survey* (1st ed.). New York, NY: Wiley.

Schneider, G.M. (2002). A new model for a required senior research experience. *SIGCSE Bulletin*, *34*, 48–51.

Scott, T.J., & Cross, J.H. II. (1995). Team selection methods for student programming projects. In *Proceedings of the 8th SEI CSEE conference on software engineering education* (pp. 295–303). London, UK. Springer-Verlag.

Sebern, M.J. (1997). Iterative development and commercial tools in an undergraduate software engineering course. In *SIGCSE '97: Proceedings of the 28th SIGCSE technical symposium on computer science education* (pp. 306–309). New York, NY: ACM.

Sharon, P., & Tuttle, M. (2001). A capstone course for a computer information systems major. *Journal of Computing Sciences in Colleges*, *16*, 41–48.

Shumba, R. (2005). Usability of rational rose and visio in a software engineering course. *SIGCSE Bulletin*, *37*, 107–110.

Sobel, A.E.K. (1996). Experience integrating a formal method into a software engineering course. In *SIGCSE '96: Proceedings of the 27th SIGCSE technical symposium on computer science education* (pp. 271–274). New York, NY: ACM.

Sobel, A.E.K. (2003). *Computing curricula – software engineering volume: Final draft of the software engineering education knowledge*. Retrieved from http://sites.computer.org/ccse/SE2004Volume.pdf

Sobel, A.E.K., & LeBlanc, R. (2002). Computing curricula-software engineering volume. Seek development and review. In *FIE '02: Proceedings of the 32nd annual frontiers in education conference*, Volume 2 (pp. F4G–2). New York, NY: ASEE.

Stansfield, S. (2005). An introductory VR course for undergraduates incorporating foundation, experience and capstone. In *SIGCSE '05: Proceedings of the 36th technical symposium on computer science education* (pp. 197–200). New York, NY: ACM.

Stein, M.V. (2003). Student effort in semester-long and condensed capstone project courses. *Journal of Computing Sciences in Colleges*, *18*, 200–212.

Streib, J.T., & White, C.M. (2002). A survey of computer science curricula at liberal arts colleges: A pilot study. *Journal of Computing Sciences in Colleges*, *18*, 36–42.

Su, H., Jodis, S., & Zhang, H. (2007). Providing an integrated software development environment for undergraduate software engineering courses. *Journal of Computing Sciences in Colleges*, *23*, 143–149.

Sun, N., & Decker, J. (2004). Finding an "ideal" model for our capstone experience. *Journal of Computing Sciences in Colleges*, *20*, 211–219.

Swift, T., & Neebel, D. (2003). Wireless, web-controlled, ball-collecting robot: An engineering and computer science cluster course. In *FIE '03: Proceedings of the 33rd annual frontiers in education conference*, Volume 2 (pp. F4E–6–11). New York, NY: ASEE.

Tan, J., & Phillips, J. (2005). Incorporating service learning into computer science courses. *Journal of Computing Sciences in Colleges*, *20*, 57–62.

Todd, R.H., Magleby, S.P., Sorensen, C.D., Swan, B.R, & Anthony, D.K. (1995). A survey of capstone engineering courses in north America. *Journal of Engineering Education*, *84*, 165–174.

Tomayko, J.E. (1987). Teaching a project-intensive introduction to software engineering. Technical Report CMU/SEI-87-TR-20, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, USA.

Towhidnejad, M., & Aman, J.R. (1996). Software engineering emphasis in advanced courses. In *SIGCSE '96: Proceedings of the 27th SIGCSE technical symposium on computer science education* (pp. 210–213). New York, NY: ACM.

Towhidnejad, M. (2002). Incorporating software quality assurance in computer science education: An experiment. In *FIE '02: Proceedings of the 32nd annual frontiers in education conference*, Volume 2 (pp. F2G-1–F2G-4). New York, NY: ASEE.

Tvedt, J.D., Tesoriero, R., & Gary, K.A. (2002). The software factory: An undergraduate computer science curriculum. *Computer Science Education*, *12*, 91–117.

van Vliet, H. (2005). Some myths of software engineering education. In *ICSE '05: Proceedings of the 27th international conference on software engineering* (pp. 621–622). New York, NY: ACM.

Vaughn, R.B. Jr. (2001). Teaching industrial practices in an undergraduate software engineering course. *Computer Science Education*, *11*, 21–32.

von Konsky, B., Hay, D., & Hart, B. (2008). Skill set visualisation for software engineering job positions at varying levels of autonomy and responsibility. In *ASWEC '08: Proceedings of the 18th Annual Australian Software Engineering Conference*. Darlinghurst, Australia: Australian Computer Society, Inc.

Waite, W.M., Jackson, M.H., Diwan, A., & Leonardi, P.M. (2004). Student culture vs group work in computer science. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on computer science education* (pp. 12–16). New York, NY: ACM.

Wallace, J.M.K., & Crow, J. (1995). Improving student information system design through evaluation and selection of an appropriate case tool. In *FIE '95: Proceedings of the 25th annual frontiers in education conference*, Volume 1 (pp. 2c3.16–2c3.19). New York, NY: ASEE.

Way, T.P. (2005). A company-based framework for a software engineering course. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on computer science education* (pp. 132–136). New York, NY: ACM.

Werth, L.H. (1987). Survey of software engineering education. *SIGCSE Software Engineering Notes*, *12*, 19–26.

William James Hall Computing Services. A quick guide to newsgroup etiquette. Retrieved from http://www.wjh.harvard.edu/wjh/newsgrp.shtml

Woodside, J.M. (2008). HOMER: Home-based object-relational medical electronic record. In *ITNG '08: Proceedings of the 5th international conference on information technology: New generations* (pp. 512–517). New York, NY: IEEE.

Žagar, M., Bosnić, I., & Orlić, M. (2008). Enhancing software engineering education: A creative approach. In *SEESE '08: Proceedings of the 2008 international workshop on software engineering in East and South Europe* (pp. 51–58). New York, NY: ACM.