Teaching a Project-Intensive Introduction to Software Engineering

James E. Tomayko
August 1987

Teaching a Project-Intensive
Introduction to Software Engineering

James E. Tomayko

The Wichita State University
and
The Software Engineering Institute
Graduate Curriculum Project

This report was prepared for the SEI Joint Program Office HQ ESC/AXS

5 Eglin Street

Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF, SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Copyright 1987 by Carnegie Mellon University.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Teaching a Project-Intensive Introduction to Software Engineering

Abstract

This report is meant as a guide to the teacher of the introductory course in software engineering. It contains a case study of a course based on a large project. Additional materials used in teaching the course and samples of student-produced documentation are also available.[1] Other models of course organization are also discussed.

1. Introduction

A first course in software engineering is a daunting experience for both student AND teacher. The students must work in cooperation with one another on a project that uses almost all their computer science skills and illustrates the techniques taught in the class portion of the course. Since this is often the most interesting single course they take, students tend to throw themselves into it at the expense of other courses. Even when they do not want to live solely for the course, the new experience of having to cooperate with their peers instead of competing with them uses unforeseen amounts of energy in communication and compromise. The instructor is also involved more heavily in this course than in most others he or she will teach. The demands of providing a sensible project or guiding students' choices, acquiring resources, giving advice and guidance, sitting in on reviews, walkthroughs, and other meetings, all contribute to a higher-than-normal time investment. However, for both student and teacher, the rewards of this course are so great as to quickly erase any bad memories of the workload. The feeling of accomplishment, the experience of learning to work together as a team, and the actual use of skills previously only exercised in limited situations all contribute to an exceptionally positive learning experience. My former students often tell me that the only college friends they keep in touch with are their teammates from the software engineering course. Their positive feelings are also reflected in significantly higher teaching ratings for the instructor.

Despite the expected positive outcomes, the beginning teacher of this course has a thousand questions and fears. How do you find a reasonable project? Should you use a "real" customer or make something up? How do you organize students into teams? What do they do? How do you keep the class meetings closely related to the project work? How should the project documents look? How do you grade teams? Will you see your spouse and children again before the semester ends? This report is aimed at answering these questions and helping both beginning and experienced teachers of the course.

The experiences described here are based on my offering an introduction to software engineering course in varied settings: to seniors in a private university, to both seniors and beginning

---

[1]An order form is located at the back of the report.

graduate students in a large state university, and to software development teams in industry. The report is organized into a short preliminary section on the various teaching models for the course, including a discussion of prerequisites, and a much longer section that is a case study of an actual course taught in the fall semester of 1986 at Carnegie Mellon to a group of seniors and first-year graduate students. The case study proceeds day by day through the course, with lesson plans for each class meeting, assignments, exams, and techniques for managing the project. Appendices contain actual documents created by the students during their project; these can serve as examples to your students. Real, industrial-grade examples would of course be better, but proprietary considerations limit what is available. The students' materials in this case are of sufficient quality to make their use reasonable as examples. Please note that they are NOT perfect. Code is not included because most instructors have a lot of experience with it, and examples are not as critical.

2. Models of the Introduction to Software Engineering Course

Beginning software engineering courses have been taught with content that ranges from no project work at all to highly-intensive project work. Figure 1 is an illustration of the spectrum of courses.

2.1. The "Software Engineering as Artifact" Model
The leftmost model illustrates the subject taught mostly by lecture, with some interaction among the instructor and students, mostly relating to questions and difficult points. There are two advantages to this approach: there is easily enough time in either a 10-week quarter or 16-week semester to present the major concepts of software engineering; and the absence of a project means that the students can concentrate on the issues the instructor wants to discuss rather than the "crisis of the week." The major disadvantage is that teaching software engineering without doing it is as bad as teaching piano playing by the lecture method. Many issues in software engineering, particularly in communication and configuration control, simply cannot be appreciated in the absence of experience. Since most projects that fail do so because of deficiencies in those two areas, we would be doing our students an injustice by not exposing them to the problems inherent in actually working on software products.

2.2. The "Topical Approach" Model
Although this is another of the all-talk, no-action models, it has the advantage of in-depth exploration of some aspects of the subject.[2] The lecture part of the course is roughly the same as the previous model, but it is supplemented by weekly presentations by the students. Each student is assigned a topic (object-oriented design, automated specification tools, verification of real-time software, etc.) and asked to read two or three papers on it and conduct a seminar for the other students. Here a succinctly written text such as one by Fairley[3] or Sommerville[4] can be used to

---

[2]This model is usually used at the graduate level.

[3]Richard E. Fairley, *Software Engineering Concepts*, New York: McGraw-Hill, 1985.

[4]Ian Sommerville, *Software Engineering,* Reading, MA: Addison-Wesley, 1985.

back up the main lectures, while the students provide their own reading material for the discussion sessions. Alternatively, essay collections such as Brooks[5] or Mills[6] can feed the discussions. Yourdon's publishing arm has also produced a pair of collections of outstanding papers in software engineering, although some of the papers are becoming dated.[7]

Figure 1: Models of the One-Semester Course

---

[5]Frederick P. Brooks, *The Mythical Man-Month*, Reading, MA: Addison-Wesley, 1975.

[6]Harlan D. Mills, *Software Productivity*, Boston: Little, Brown, 1983.

[7]Edward Yourdon, ed., *Classics in Software Engineering*, New York: Yourdon Press, 1979, and Edward Yourdon, ed., *Writings of the Revolution: Selected Readings on Software Engineering,* New York: Yourdon Press, 1982.

2.3. The "Small Group Project" Model

This model includes a project as part of the course for the positive reasons discussed above. Currently the most common model of this course, it makes a fairly even division between project work and class work. The projects chosen are often familiar to the students and can be done in a single term. In Kant's presentation of this model,[8] she lists some suggestions: a string-handling package for standard Pascal, a reservation system for the computer center's terminal room, and an interactive text editor, among others. Typically, from three to five students are included in each team. This model provides the students with some of the experience needed to apply the software engineering concepts discussed in class. However, it is deficient in its ability to enable the students to experience the critical difficulties inherent in doing "programming-in-the-large." The use of a small project, small teams, and often fictitious customers simply extends the "programming-in-the-small" experience normally gained in computer science course work. However, adequate attention to configuration management and quality assurance issues can be obtained in this model by having teams act out those roles for each other. Doug Tygar at Carnegie Mellon ran a version of this model with such additions during the spring of 1987. His experiences are summarized in Appendix Z.

2.4. The "Large Project Team" Model

This final model posits that the best way to learn techniques for dealing with programming-in-the-large (which often means dealing with programming-in-the-many) is to conduct a large team project within the class. Quite simply, there is one project, usually one piece of deliverable software; often there is also a real customer who provides a form of motivation different from grades. If we accept that a central objective of the course is to immerse the student in a practical, real-life software product development process, then this is how to do it. The figure indicates, quite correctly, that the majority of the work the students do is on the project. Many instructors object to this model because they feel that it is too difficult to manage. Not so. If it is too difficult to manage a project team of 15 to 30 students, then how is it possible to manage a real-life corporate development team of 15 to 30 software engineers? I have run introductory courses using this model a half dozen times, in general quite successfully. Even with partial failure, the students learn more about real software engineering than with any other model. The key to using this model is to remember the following:

THE STUDENTS ARE DOING THE PROJECT. YOU ARE NOT. YOU ARE MANAGING THE PROJECT, WHICH MEANS THAT YOU ARE DELEGATING NEARLY ALL ASPECTS OF THE PROCESS TO THE STUDENTS.

The remainder of this report is a case study of exactly how to do this.

---

[8]Elaine Kant, "A Semester Course in Software Engineering," *ACM Software Engineering Notes,* Vol. 6, No. 4, August, 1981.

3. Prerequisites

If this course is to be taught to undergraduates, the nominal prerequisites include:

- Close familiarity with at least one and preferably two high-level structured or structurable languages, such as Pascal or Ada[9] and Cobol or Fortran
- Facility with assembly language
- Grounding in the fundamental principles and algorithms of computer science such as data structures, recursion, sorting and searching, and so on
- Basic knowledge of computer architecture
- College-level writing courses
- Senior standing.

The last is added to ensure that students have some high-level computer science classes, such as theory, compiler design, and survey of languages and have achieved some maturity in both the subject and in life. Jon Bentley once said that a useful prerequisite for a graduate software engineering program is that the student "should have been married at least once." We will settle for less, especially since the entire computer science curriculum up to this point provides tools for software engineering but little context.

4. The Case Study: A Highly Project-Intensive Introduction to Software Engineering

4.1. Presentation
To stress that the class work involved in this course directly complements the project work, the approach of this case study is to go step by step through the process of preparing for the course, teaching the individual classes, and coordinating with the project pretty much as outlined in the syllabus. We will describe the activities needed at each step, along with support materials, assignments, exams; and we provide annotations and discussions of specific points.

Note that this case study is of a one-semester course.

4.2. Preliminary Preparation
Getting ready to teach this course begins several months in advance. You need to determine class size, choose a project, select the development environment (including tools), and develop the syllabus. It is critical that these factors be decided *before* the first day of class, because efficient launching of the project is critical to the overall success of the course.

---

[9]Ada is a registered trademark of the U. S. Department of Defense, Ada Joint Program Office (AJPO).

4.2.1. Class Size

A class size of 15 to 20 students is enough critical mass to cause the communications and configuration control explosion that you are trying to engineer. Actually, any number over that helps you because it increases the entropy. However, to be realistic in terms of helping students, grading papers, and general management, 30 to 35 ought to be the absolute upper limit. Any more than that and you compromise your ability to do your job. Using assistants often backfires since they also require management. Frankly, the main reason for keeping the class size limited to 20 or so is that is all you need to successfully do a project that has a four-month development period. In this case study, the actual number of students that showed up the first day was 38. By the next class, eight had disappeared, presumably because they had read the syllabus (another good reason for handing one out right away). The remaining 30 were too many for the actual project, so I decided to develop a single set of requirements, then do a dual implementation in Pascal and Ada. This meant that some roles would be duplicated (I needed coders for both languages, for example), thus using up some manpower.

The reason I place class size as part of "preparation" is that you need time to fight with the department head or dean to gain a size limit for your course. Please feel free to reference the last paragraph. In fact, tell them to call me up if you want. Failing to obtain a limit, you must then use the pre-registration information to think about how big a project you need, or how you are going to restructure the one you have.

4.2.2. Choosing a Project

In a project-intensive course, it is very important to come up with an interesting project. This, to my mind, immediately eliminates the commonplace stuff like text editors, parts of operating systems, text editors, reservation systems, and text editors. Software engineering, though often tremendously exciting, has long stretches of tedium associated with it. A gripping project sustains and motivates the students. However, if the project is for a real customer, then that factor compensates somewhat for a little somewhat. For example, some of my more recent project choices include an automated scouting analysis system for an NCAA Division I football team, a choreographer's assistant, a simulation of the Gemini spacecraft onboard guidance computer (including applications software), and a mission planning simulator for a manned research station on Mars. The weakest project of these in terms of general interest was the choreographer's assistant, but the constant interaction with computer-naive dancers dependent totally on the students balanced things out. Also note that each of these projects was unlikely to have "experts" in the class that outshone the other students in requirements analysis. I had expected to have a least a couple ex-high school football players in the first course mentioned above, but in reality none enrolled. There was one ex-dance student in the second course, but the space flight-related projects had no one with any prior experience. Choosing a project area in which students do not have much experience creates a situation where the students have to interact with the user very intensely during the requirements definition stage, thus developing a bond that can be used for motivation throughout the project. I can still remember my department head's amazement when she looked in on my software engineering lab section breaking down a football game film with the coaches.

So where are these fascinating projects? Everywhere. Non-profit agencies need all kinds of help; many engineering and science research groups on campus could use computational tools, and there are at least 432 football teams out there using manual scouting analysis methods. Many of these projects can be repeated after a suitable interval. When coaches are fired, the new coach almost always has a different offensive and defensive philosophy, which calls for a totally new scouting system. Research projects run out of funds and are replaced by new ones with similar requirements. There is nothing wrong with repeating a similar experience, just as long as it is new to your students.

The project used in this case study is a simulator of the activities of a manned Mars research station used to do mission planning and analysis. The idea for the project grew out of a discussion I had with Dr. Chris McKay of NASA's Ames Research Center. Chris is the volunteer coordinator of the Planetary Society's Mars Institute, which conducts educational conferences and supports college courses dealing with exploration of the planet Mars. Chris had observed the use of a simulator for modeling Space Station operations at Ames. He and I thought that creating a similar simulator for a Mars Station would be of interest to both the students and the team developing the Space Station simulator. He arranged for me to obtain copies of the documentation for the Space Station simulator and some Mars Institute reports relating to mission profiles. I also obtained a report on Manned Mars Missions from NASA's Marshall Center. By extracting relevant materials from these three sources, I was able to assemble a package (reproduced as Appendix A) that would serve as a basis of information for the students in creating the actual requirements.

4.2.3. Choosing the Development Environment

After you choose the project and know your class size, the next preliminary step is choosing the development environment. Sometimes this is a trivial matter, since only one environment exists. If PCs or mainframe systems are all you have available, then you might have to adapt the project. If PCs are available but the class size dictates that they will be overused, then you might have to stay on the mainframes. Sometimes the customer will have a specific target machine. For instance, the football team we did the scouting system for had a booster donate an IBM System 23 with only BASIC. My classes have used PCs, mainframes, MicroVAX workstations, and super-minis, all without having to abandon good software engineering practice. Furthermore, remember that only a very small percentage of the class will actually be coding. During the choreographer's assistant project, we were limited to developing code on two graphics-equipped PCs, which was no problem. Project documentation, however, had to be maintained on a central mainframe system.

A more influential consideration is the availability of tools. You will need, at least, an editor, an appropriate compiler, version control tools, and electronic mail and/or bulletin board facility. The first two are familiar to everyone and need no elaboration, although the availability of a particular compiler might limit the choice of machines. Configuration management and version control tools such as RCS under UNIX, CMS under VMS, and Softool's CCC under VM/CMS, provide most of what is needed. Without these tools, configuration management must be done manually. The sample configuration management plan in Appendix B describes version control in a PC development environment. The e-mail or bulletin board is extremely useful. The main difficulty in keeping

this course realistic is that the students are not in the same general area all day as they would be in a software development company. Bulletin boards and e-mail help overcome that problem. When I taught the case study course, my office was in the Software Engineering Institute about a mile from campus. Unless they were willing to walk the mile over and back, the students were limited to class time and my twice-weekly two-hour office hours on campus to see me face to face. As a result, most of the development information and out-of-class instruction was conducted by electronic mail. I handled about 400 messages relating to the course during the semester. Fortunately, most mainframe and minicomputer systems now come with some sort of communications. However, non-networked PCs are obviously not capable of being used for mail. That is another reason to keep documentation and mail on some central system even if the target is a PC.

4.2.4. Developing the Syllabus

After you know what the class size will be, what project you will be doing, and where you will be doing it, it is time to develop the syllabus. In this section we will walk through the case study syllabus in Appendix C to discuss the various considerations. Note that the case study was taught as a three-semester-hour course with class meetings of 80 minutes twice a week. I have also taught the course with two hours of lecture and two hours of laboratory time scheduled per week. I found the latter a less demanding schedule from the instructor's standpoint.

I will take a top-down approach to explaining the example syllabus:

Header: The heading combines the symbols of the Planetary Society report on a manned Mars research station and the Society itself. This helps set the theme of the project in the minds of the students. In the case of the Gemini computer simulator, I used the same header McDonnell-Douglas put on all Gemini documents. In the case of the football software, I found a "Doonesbury" panel in which Zonker is explaining to the football-playing character B.D. that to be drafted into the pros he had to learn about computers: "Even the linemen use pocket calculators."

Housekeeping: Below the header is the normal "where to find me" information, doubly important for this course, so students don't hold up work waiting for someone to find you and ask a crucial question. Besides, it is good practice to advertise your availability; it makes the students more comfortable.

Textbooks: Fairley's text contains almost enough information for a project-intensive course, with some deficiencies easily moderated by handouts. The book's spare approach is justified by excellent references. Brooks is the classic of the field, simply because it shares principles derived from experiences few people will have but which they can apply in their current situation. Neither book is read in order from front to back, because their topics are not in the precise order necessary to support the project.

Evaluation: Half the students' time, and subsequently half the grade, resides in the project. Many instructors worry about how they are going to assign an individual grade to a student immersed in a group. My method is to make assignments sufficiently well delineated that individuals are responsible for one thing or another. When I discuss the bi-weekly project assignments, you will

see this in action. The remaining 50 percent is divided between exams and a "discretionary fund" called "participation." The purpose of this last segment of the evaluation is to provide artificial motivation for participating in class discussions and getting along within the groups. It also leaves the instructor some subjective room to make final decisions between As and Bs.

Schedule: The schedule of readings and class activities is carefully constructed to support the project so that the project can reinforce the readings and presentations. Therefore, it is important when building the syllabus to schedule items according to some priority order:

1. Exams: Grades are almost always due at mid-term and at the end of the term, so the exams are scheduled first. Note that both hour exams are given in class. The final exam time, usually three hours in length, is set aside for a final presentation of the substance of the project. It is open to the public and contributes to the "participation" grade.

2. Project milestones: The more time students spend on requirements analysis, design, and preparation for testing, the less time they need for coding and integration. Therefore, one month is devoted to the development of requirements and functional specifications, with milestones two weeks apart. Design is scheduled for a month, with preliminary and detailed design milestones two weeks apart. Coding is given two weeks, followed by a month of testing and validation. Milestones are marked by in-class walkthroughs and reviews of all or part of the documents. These give you an opportunity to demonstrate walkthrough, review, and inspection techniques. During the time this case study was taught, I was involved in Software Engineering Institute activities that required my absence for two one-week periods. I purposely scheduled my absences during the design and testing stages to give the students that much more time to work. I have done this other semesters as well, and it usually helps keep the project on schedule.

3. Class presentations: Lectures and discussions are scheduled last. It is barely possible to stay one half of a life-cycle stage ahead of the project. The list of topics on this syllabus is ambitious, but quite attainable. Each day of the course is described in detail below, but here is a brief description of the content of each class topic:

   - *Software Engineering: Programs as Products/Life-Cycle Models:* Introduction to the concept of software engineering as opposed to computer science or programming. Discussion of software as products to be used by other than the developers. Presentation of different life-cycle models, such as waterfall, rapid prototype, incremental development, etc.

   - *Development Standards/Project Organization:* Standards for software development, including government standards, IEEE standards, and corporate standards. Models of team organization, such as surgical team, democratic, and chief programmer.

   - *Requirements Engineering:* How requirements are determined. Interactions with customers, marketing, and development organizations. Stating requirements and developing the requirements document.

   - *Controlling Disciplines: Quality Assurance (QA) and Configuration* Management: What software QA and configuration management organizations do in a software project. Relationship of their activities to the developers. Concept of independent verification and validation.

   - *Cost, Size, and Manpower Planning:* These topics relate to project manage-

ment. Cost-estimation techniques and methods such as COCOMO. Software size estimating and its relation to schedule and cost. Manpower loading on a project over the life cycle. Mythical man-month discussion.

- *Specification Techniques:* Formal specification tools and techniques such as on-line tools and data flow diagrams. Functional specification development.

- *Design Concepts and Methods:* Survey of design methods: top-down structured, Jackson, Warnier-Orr, object-oriented, etc. Advantages and disadvantages of each in differing problem domains.

- *Design Representation:* Using structure charts, HIPO charts, data flow diagrams, pseudocode, and other tools in implementing designs.

- *Structured Programming and Implementation Considerations:* Review of the concepts of structured programming. Discussion of Bohm and Jacopini paper.[10] Applying structure to non-structured tools such as assembly languages. Coding considerations in FORTRAN and COBOL versus Pascal and Ada.

- *Software Testing and Integration Concepts:* Unit testing techniques, white-box versus black-box testing. Concept of coverage. Integration methods, such as top-down, bottom-up, and Big Bang. Development of testing and integration suites.

- *Verification and Validation:* Formal verification, concept of validation, and acceptance testing. Development of validation suites. Automated testing.

- *Post-Development Software Evolution:* The software maintenance problem. Designing for maintenance. Developing a maintenance handbook for a software product. Reverse-engineering of software product documentation to improve maintainability of existing code.

- *User Documentation:* Characteristics of good user documentation. Writing user documentation if you are a developer. Document organization and style; ways to assist the reader.

With the completion of the syllabus, your preliminary preparation for the course is finished. Now it is time to enter into the day by day activity of running the class.

4.3. The Course Day by Day

This subsection presents, for each class meeting, the objectives, activities, and assignments. The portions in helvetica text are the lesson plans I wrote when I taught the course. I added the italicized parts when preparing this report to expand on the rationale for the various items, or to clarify their use. Each day's plan is headed by the number of the class meeting and the actual date on which it occurred for reference to the syllabus.

---

[10]C. Bohm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," in the *Communications of the ACM*, Vol. 9, No. 5 (May, 1966), pp. 366-71.

4.3.1. Class #1: Introduction, 28 August 1986

Objectives:

1. Establish that software engineering is distinct from computer science.

2. Explain the context of software engineering education.

3. Differentiate between programs and programs that are products.

4. Present the concept of the life cycle, including the waterfall, rapid prototype, and incremental release models.

5. Introduce the class project.

Activities:

1. Ask for definitions of science and of engineering.

2. Show how science and engineering are related (chemistry and chemical engineering, for example). Discuss the relationship between computer science and software engineering.

3. Read a disclaimer from a typical software product to illustrate how software is still not quite engineered. *Sample disclaimer is in Appendix D. I have found that reading a disclaimer aloud, followed by rephrasing it as a disclaimer for an automobile, is very effective.*

4. Discuss the role of the course in considering programs as products.

5. Use Brooks' chart of the effort relationship between programs, products, programming systems, etc. (p. 5 of *The Mythical Man-Month*) as a springboard to making a distinction between programming-in-the-small and in-the-large.

6. Distribute a chart of the waterfall life-cycle (*See Appendix E*) explaining the relationship between the steps and stressing the iterative nature, deliverables, and control mechanisms.

7. Expand the life cycle discussion to include rapid prototyping and incremental release models.

8. Review syllabus, stressing that readings are to be done in advance and describing the evaluation procedures.

9. Introduce the class project as follows (final 30 minutes):

   - Show the Mars Mission video tape. *I was able to obtain a six-minute video from NASA showing a typical mission profile.*

   - Briefly describe the project, which is the OpSim for the Mars Research Station.

   - Describe the concept of roles; hand out the roles sheet *(see below)* and the Mars information *(Appendix A)*. Explain that roles for the project will be chosen during the next class.

   - Review the tape again to expand upon Mars mission profiles. *Here I discussed various considerations relating to optimal trajectories versus time spent on Mars before return.*

Assignment:

1. Hand out standards tables of contents. *These are in Appendix F. They are from NCR, Boeing, and Military Standard 2167. The intent is to have something to use as a basis for discussing development standards during the next meeting.*

2. Reminder: read assigned readings. *These were Fairley, pp. 1-24 and 39-53, plus Brooks, pp. 3-9 for this class, and pp. 53-61 of Fairley and pp. 29-58 of Brooks for the next meeting.* Also read standards tables of contents.

3. Read project materials and roles sheet. Have a good idea of what you want to be.

To save time, the possible roles a student can take in the project were described in a handout so that the students could think about their possible jobs for the semester. The roles sheet is reproduced on the following page.

Roles for Class Project

Principal Architect: Responsible for the creation of the software product. Primary responsibilities include authoring the requirements document and specification document, advising on overall design, and supervising implementation and testing.

Project Administrator: Responsible for resource allocation and tracking. Primary responsibilities are cost analysis and control, computer and human resource acquisition and supervision. Collects data and issues weekly cost/manpower consumption reports and the final report.

Configuration Manager: Responsible for change control. Primary responsibilities include writing the configuration management plan, tracking change requests and discrepancy reports, calling and conducting change control board meetings, archiving, and preparing product releases.

Quality Assurance Manager: Responsible for the overall quality of the released product. Primary responsibilities include preparing the quality assurance plan, calling and conducting reviews and code inspections, evaluating documents and tests.

Test and Evaluation Engineer: Responsible for testing and evaluating individual modules and subsystems and for preparing the appropriate test plans.

Designer: Primary responsibility is developing aspects of the design as specified by the Architect. During the pre-design stage, this person could assist in a literature search to explore similar products or problems.

Implementor: Primary responsibility is to implement the individual modules of the design and serve as the technical specialist for a particular language and operating system. During the requirements specification and design stages, the implementors could develop tools and experiment with new language constructs expected to be needed in the product.

Documentation Specialist: Responsible for the appearance and clarity of all documentation and for the creation of user manuals.

Verification and Validation Engineer: Responsible for creating and executing test plans to verify and validate the software as it develops, including tracing requirements through specification, design, coding, and testing. Also responsible for code inspections. Acts as a member of an independent group.

Maintenance Engineer: Primary responsibility is creating a guide to the maintenance of the delivered product.

Obviously, the students cannot have had enough experience to really know what each of these roles involves. Equally obviously, you will not know enough about the strengths and weaknesses of each student on the first day to be able to make realistic role assignments. Therefore, it is best to allow self-selection on the part of the students. That way they have no one to blame for a bad choice but themselves!

4.3.2. Class #2: Standards and Organization, 2 September 1986

Objectives:

1. Introduce the concept of development standards.

2. Discuss the various methods of organizing programming teams.

Activities:

1. Pass around a sheet for names and e-mail addresses. *This enabled me to know how to contact each student individually.*

2. Review the life cycle by focusing on requirements, specs, design, code, and testing. *In other words, accentuate deliverables as a lead-in to standards.*

3. Ask: How do standards relate to the life cycle?

4. Introduce the concept of standards by showing The WSU CS Department programming standard. *I passed around an example of a simple departmental programming standard used at The Wichita State University. See Appendix G.*

5. Discuss corporate standards, government standards, IEEE standards.

6. Ask: What was Brooks' metaphor for a programming team? Discuss how expertise is interrelated. *This is the discussion of a surgical team.*

7. Discuss chief programmer teams. *I have found Mills' book of essays useful preparation for this.*

8. Discuss democratic teams.

9. STRESS THAT EVEN THOUGH THERE IS A DIVISION OF LABOR, EVERYONE SHOULD HAVE A CONCEPT OF THE PRODUCT.

10. Introduce the organization for the project.

11. Assign project tasks. *Here is where I ask for volunteers and resolve differences. As discussed above, due to the size of the class, I separated the students into Ada and Pascal development teams working from a single requirements document. The actual breakdown is reproduced below.*

Assignment:

Read Brooks, pp. 61-69, and the Viking Document Handout. *See Appendix H.*

The actual breakdown for teams in the course started out like this:

Mars Research Station OpSim
Development Organization

Principal Architect: Chris Fedor   fedor@y.cs.cmu.edu

	ADA TEAM		**PASCAL TEAM**	
Admin:	Eric Borm	EB0P@TB	Steve Gale	SG0Q@TC
CM:	Walter Smith	wrs@k	Jon Lange	JL13@TC
QA:	Claus Cooper	CC2W@andrew	Mike Blackwell	mkb@rover.ri
Design:	Phillip Aspenwall	aspen@andrew	Ruth Matsumura	RM29@TD
	Mike Gillinov	mdg@g.cs.cmu.edu	Mike Swatko	MS3E@TE
	Jochen Knieling	JK1R@TC		
Code:	Doug Bunting		David Johnson	DJ05@TD
	Steve Hecht	hecht@s.cs.cmu.edu		
	Peter Psycharis	PP08@TD		
T&E:	Sanjay Agrawal	agrawal@me	Jeff Lynn	JL0V@TC
	Mark Abramowitz	MA04@TB	Todd Ihrig	TI02@TC

Documentation:	Daniel Bernstein	DB2Q@TB
V&V Team:	Renaud DeBarbuat	RD1M@TC
	James Barton	
	Frits Habermann	jfh@y.cs.cmu.edu
	Rich Wonsonegoro	RW1F@TF
	Rob Kedoin	RK15@TC
	Brad Willitts	BW0L@TD
Contract Monitor:	James E. Tomayko	jet@sei.cmu.edu

There were two late adds, Marc Scheurer and Magnus Kempe, both visiting students from Europe. They had extensive Ada experience, so they joined the Ada development team as coders to help compensate for the lack of Ada knowledge in the group. Note that only one Pascal coder was needed.

The first set of project assignments was also distributed on the second day. Where the word "paper" appears, "chapter of a book" may be substituted.

Roles for Class Project: Due 16 September 1986

Note: Everyone MUST keep track of the hours spent on project activities. Turn in weekly activity report forms to appropriate project administrator.

This is a tool that you can use to keep track of the effort everyone is making, as well as train project administrators. By reviewing log sheets, it is possible to tell what percentage of effort is being expended on the project versus class work. This helps you to fine-tune the assignments as the course progresses.

Principal Architect: Write requirements document. *As the customer's representative, I was available to answer questions about the customer's needs and wants, but the actual document was prepared by the students based on their own ideas and on the materials handed out on the first day of class.*

Project Administrator:

* Create weekly activity report forms (by Thursday). *This form is reproduced in Appendix I. Note that it contains provisions for tracking both PC and mainframe computation costs. The project administrators had to determine the prices for PCs; the costs for mainframe usage was indicated to the students as they logged off. The project administrators handled all interaction with the Computing Center, and the students were warned not to use their Software Engineering accounts for any other classes, as that would corrupt cost tracking.*

* Collect and analyze labor costs for first two weeks. *Appendix J is the final cost-breakdown report for the project. Intermediate reports were similar except that they contained projections where the actual figures are printed in this case.*

* Project labor and machine-time costs for the length of the project.

* Make recommendations for computer budgets for all team members.

Configuration Manager: Prepare configuration management plan [adapt IEEE Standard].

Quality Assurance Manager: Prepare quality assurance plan.

Test and Evaluation Engineer: Ada team: Locate papers on testing Ada programs, and derive unique characteristics, if any, of testing Ada. Prepare a written report evaluating what you have learned. *Note that this assignment and the following ones get team members who do not yet have actual project work to do involved in training.*

Pascal team: Locate two papers each relating to the topic of *unit testing*; summarize them in a written report to be made available to the Ada team.

Designer: Ada team: Study at least two papers on the subject of *object-oriented design*. Summarize the method in a written report, and present an oral report to the principal architects the evening of 16 September.

Pascal team: Study at least two papers relating to *data flow techniques*. Summarize your findings in a written report and make it available to the Ada team.

Implementor: Ada Team: Locate and study books explaining the Ada language. Concentrate on the concepts of *package* and *task*. Summarize the way both structures are implemented in the language, and make the report available to the Ada implementors, designers, and architects.

Pascal team: Prototype screen display subroutines. Submit screen designs and the resulting code. *In the past I have found this to be the most time-consuming aspect of implementation, so I wanted students to do some rapid prototyping early. In the end, the system was implemented in batch form, so the screens were not needed --- a nice touch of realism!*

Documentation Specialist: Work with the architects in drafting and finalizing the requirements document. Study and evaluate three user manuals and write a critique.

Verification and Validation Engineer: Locate papers on the subject of *software inspections*, read them, and summarize what you have learned in a written report.

Maintenance Engineer: Locate two papers on the subject of software maintenance, read them, and summarize what you have learned in a written report. *Due to the number of students and the split into dual teams, this position was never filled.*

4.3.3. Class #3: Requirements Engineering, 4 September 1986

Objectives:

1. Understand the concept of traceability.

2. Understand the concept of not rushing the implementation considerations.

3. Differentiate between *requirements* and *specification*.

4. Discuss methods of extracting requirements.

5. Outline a requirements document.

Activities:

1. Introduce the concept of *traceability* by using the Shuttle screens example. *In Appendix K, there is a screen layout taken from the Shuttle onboard software requirements document prepared in the mid-1970s. The bottom screen is a "snapshot" of the same screen as it appeared years later. The point is that the actual implementation of the screen can be traced back to the requirements.*

2. Continue with traceability and discuss separation of requirements and implementation using JPL Viking example. *Appendix H, handed out at the end of the last class, is an excerpt from the software design and implementation of the Viking Orbiter Command Computer Subsystem. The excerpt is of the Command Loss routine that essentially acted as a watchdog timer. The point again is to show traceability.*

3. Define requirements as statements of *what* a product must be capable of doing.

4. Discuss how requirements can be extracted: *I have found it most effective to develop this list in a brainstorming session with the students, rather than just listing them on the chalkboard.*

    - Interviews
    - Observation of the activity to be programmed
    - Study of existing documentation
    - Paper and pencil prototypes
    - Software prototypes

5. Outline of typical requirements document:

    - General introduction
    - Inputs
    - Constants
    - Processing
    - Outputs
    - Performance characteristics
    - Host characteristics
    - Exception handling

Assignment:

Read the Elevator Example, *Appendix L*, and QA handouts, *Appendix M*, plus Fairley, pp. 311-322.

4.3.4. Class #4: Controlling Disciplines, 9 September 1986

Objectives:

- Demonstrate how quality assurance and configuration management are important to software projects and how they act as controlling disciplines.

Often this subject is left to the end of the course, if it is treated at all (note how late Fairley gets to it). This is a grave error, because it gives the message, all too often reflected in real projects, that these disciplines are afterthoughts. It is crucial to software quality to have a clear idea of what the configuration management and quality assurance procedures will be from the beginning of the development process. More information on configuration management is obtainable in the SEI Curriculum Module SEI-CM-4-1.3 [1] and its associated Support Materials Package [2].

Activities:

1. Draw diagram showing product integrity in the center, project management coming from the top, QA, SCM, and V&V from the left, and the development disciplines from the right. Explain the relationships.

2. Discuss why CM is needed from the beginning of a project (as opposed to where Fairley puts it in his book). Use story of the company which put CM back to the release date and then couldn't understand why nothing got done on time. *This story is from a company I actually worked with. CM was not instituted until release; this meant that any change requests prior to release were accepted wholesale. The project managers could not figure out why their schedules were always wrong!*

3. Define configuration items, show relationship to baselines.

4. Differentiate between discrepancies, changes, and enhancements, using the examples in the handout (*Appendix L*).

5. Trace the change evaluation process (control board composition, what to evaluate, what to do when it has been evaluated).

6. Discuss version control and tools, such as RCS.

7. Evaluate CM plan for the project.

8. Define QA.

9. Brainstorm QA functions.

10. Discuss QA questions in reading (*Appendix M). The purpose is to help the students understand the differing interpretations of the QA role dependent on the developing organization and outside standards.*

11. Discuss QA plan for our project.

Assignment:

Read Fairley, pp. 64-84, and Brooks, pp. 13-26 and 87-103.

4.3.5. Class #5: Cost, Size, and Manpower, 11 September 1986

Objectives:

- Review software cost-estimating techniques and the pitfalls that accompany them.

Activities:

1. Ask: Why is the mythical man-month mythical? Elicit these answers from students:

    - It is a fallacy that all will go well.

    - It is an assumption that men and months interchange.

    - Managers are not stubborn enough to develop realistic estimates and stick to them.

    - Progress is poorly monitored.

    - Usual solution to schedule slips is to throw more bodies into the fire.

2. Discuss reasons why man-months do not work; often development is sequential (cannot test what you have not coded; use pregnancy as an example), training and communication hinder things.

3. Discuss Brooks' partitioning of total project time, relate to Apollo program. *It spent 80 percent of the total time in testing!*

4. Read "Gutless Estimating" on p. 21 out loud and discuss.

5. Look at chart of difficulty on p. 91.

6. Discuss Fairley's list of cost variables:

    - Programmer ability (5-1 ratio, large projects have more turkeys because of simple probability and because they are never managers; this does not happen only in programming, as secretaries probably are 5-1 or worse.)

    - Product complexity (applications, utility, system). Discuss p. 67 chart. What is Boehm's key factor in equations that can screw up everything? LOC.

    - Size.

    - Available time. Extending time cuts costs.

    - Required reliability.

    - Level of technology (assembler versus high level; quote from Brooks on this; tools and techniques; this is why Ada may help.).

7. Discuss cost estimation techniques:

    - Expert judgment

    - Delphi

    - Work breakdown

    - COCOMO

8. Present TOMAYKO'S LAW: Make your best time and cost estimate, double both, and never back down.

9. Staff estimation: Discuss variations in loading across the life cycle.

Assignment:

Principal architectA gives brief review (20 minutes) of requirements document and approach; then we go over things line by line. PICK UP YOUR COPIES AT THE ENGINEERING LIBRARY AFTER 1600 ON MONDAY.

4.3.6. Class #6: Requirements Review, 16 September 1986

Objectives:

- Demonstrate walkthrough-type review, using the requirements document.

Activities:

1. COLLECT ASSIGNMENTS. *These included the requirements document (Appendix N), the Configuration Management Plan (Appendix B), and the Quality Assurance Plan (Appendix O). In addition, the configuration managers prepared discrepancy and change reporting forms (Appendix P), and the quality assurance personnel released both documentation (Appendix Q), and source code (Appendix R), style standards, and guidelines. See the comments below. The other students turned in their individual assignments.*

2. Conduct walkthrough-type review of requirements document led by principal architect. Discuss the difference between this and an inspection. *This review started the students evaluating the work of the principal architect in light of what they themselves understood of the problem. A quite lively exchange ensued.*

Assignment:

Hand out new two-week assignment sheet for project. Next class: Read Fairley, pp. 88-132.

Some comments on the documents submitted at this class meeting:

Both the quality assurance and configuration management plans were adapted from outlines published in IEEE Standards. Both plans turned out to be extremely successful in guiding the team members in what to expect from CM and QA and what procedures to follow when reporting and evaluating discrepancies and changes. The supplements on document and source code preparation were invaluable in creating sufficient conformity to make the review and inspection process easier. Copies of these documents were distributed to the class members and kept on-line in a public account.

The requirements document is laid out in such a way as to contribute to traceability later, with numbered paragraphs and clear divisions.

NOTE THAT ALL OF THESE DOCUMENTS CAME UNDER CONFIGURATION CONTROL WITHIN 48 HOURS AFTER REVIEW, THAT TIME BEING USED TO MAKE OBVIOUS CHANGES GENERATED IN THE REVIEW PROCESS. ANY FURTHER CHANGES HAD TO BE CONSIDERED BY THE CONFIGURATION CONTROL BOARD.

The new assignment sheet was:

Roles for Class Project: Due 30 September 1986

Principal Architect: Write specification document. *The intent was to force the architect to think in terms of functionality.*

Project Administrator: Prepare new cost projections and analysis of two-week block. Continue to control computer budgets.

Configuration Manager: Revise CM plan, if necessary. Conduct your business as per CM plan. *I reviewed and made suggestions to both this plan and the QA plan.*

Quality Assurance Manager: Revise QA plan, if necessary. Conduct your business as per QA plan.

Test and Evaluation Engineer: *More training.*

Ada team: Test Ada display screens. Prepare test plan based on Pascal version, and implement it. Results go to QA for analysis.

Pascal team: Prepare test plan of Pascal screens, and implement it. Results go to QA for analysis.

Designer: Continue to support the principal architect. Conduct preliminary discussions of design strategy WITHOUT THE PRINCIPAL ARCHITECT. Mail minutes to me. *I was trying to force the designers to work from the requirements, rather than from direct interaction with the person.*

Implementor: *More training.*

Ada team: Write an Ada program to calculate the squares, cubes, and square roots of the numbers from 1 to 100, using three separate modules for each operation.

Dave, Magnus, and Marc: Translate Pascal screens to Ada. Deliver to Ada T&E soon enough for test.

Documentation Specialist: Work with Vijay Reddy, VR05@andrew, to begin blocking out user manual. *I arranged for a technical writing major to assist the documentation specialist. This student was taking a senior-level tech writing course, and we became his project.*

Verification and Validation Engineer: This team should meet and organize itself. It needs a team leader and some other organization, which is left to you. Send me minutes. Discuss approaches to verification based on the requirements document. *This team was so large that I wanted it to have an internal organization and also a single entry point to simplify my task. This early granting of autonomy resulted in a very effective team.*

4.3.7. Class #7: Specification Techniques, 18 September 1986
Objectives:

1. Familiarize the students with the content of a specification.

2. Familiarize the students with formal specification tools.

3. Familiarize the students with automated specification tools.

Activities:

1. Review Fairley's table of contents for a specification and evaluate what goes into the various parts.

2. Review the formal tools, emphasizing that they must be studied in more detail than is presented in the text before they can be used.

3. Review SADT, SREM, and the other automated tools, stressing common features and limitations.

Assignment:

Read Fairley, pp. 137-152.

As a general note for this meeting, the absence of automated specification tools in our development environment (and nearly everywhere else in the academic community) made this presentation somewhat shallow.

4.3.8. Class #8: Design Methods I, 23 September 1986

Objectives:

1. Differentiate between external and internal design.

2. Differentiate between architectural and detailed design.

3. Discuss design concepts: abstraction, information hiding, structure, modularity, concurrency, verification, aesthetics.

4. Define coupling and cohesion.

Activities:

1. Discuss how design and coding are better understood than specification and testing.

2. Separate design into external and internal. External relates to input/output. Internal is architectural and detailed.

3. Discuss abstraction in terms of functional, data, and control abstractions.

4. Present information hiding in terms of well-defined interface development.

5. Discuss structure.

6. Present the concept of modularity, discuss how to decompose into modules, size criteria, and reuse criteria.

7. Discuss using concurrency as a design concept.

8. Present the concept of verification in terms of the design.

9. Discuss the concept of aesthetics.

10. Discuss coupling: complexity of interfaces (use of globals, etc.)

11. Discuss cohesion: i/o, initialization modules, module "sticks together."

Assignment:

Read Fairley, pp. 161-188.

Most students will have been exposed to most of the material in this class meeting sometime prior to taking this course. This presentation focuses that knowledge in terms of design. The distinction between external and internal design might be new.

4.3.9. Class #9: Design Methods II, 25 September 1986

Objectives:

1. Present the concept of a *design method*.

2. Introduce the following design methods:

- stepwise refinement

- structured design

- integrated top-down development

- Jackson

- object-oriented design

Activities:

1. Discuss how a design method is sometimes institutionalized and why it should not be. *In some organizations a particular design method is adopted and used in all instances. Except in companies that concentrate on one type of product to the exclusion of all others, this is an error, as different methods work better in different domains.*

2. Using the Gemini simulator problem as an example, demo the various characteristics of the design methods. *I used an example of writing a software simulator of the Gemini spacecraft guidance computer, which is essentially a real-time system. Its specification is Appendix S. Here simply use any example that is very familiar to you to demonstrate the differences in the different design approaches.* For instance:

- Stepwise refinement: show how high-level statements can be refined into lower level ones.

- Structured design: develop a structure chart and show interfaces.

- Integrated top-down development: "design a little, code a little, test a little"; this is a life cycle approach as well.

- Jackson: let structure of the problem dictate structure of the solution.

- Object-oriented design: use report done by Ada group as a springboard.

Assignment:

Move Fairley, pp. 181-188, to next design assignment. *A little fine tuning here...*

Next class: Specification review, run by QA. *QA said in its plan it would run the reviews, so I stayed out of it.*

Exam next Thursday. *Reminder.*

4.3.10. Class #10: Specification Review, 30 September 1986

Objectives:

- Conduct a specification review, showing the differences between requirements and specification.

Activities:

- COLLECT ASSIGNMENTS.

Assignment:

Study for Exam One.

The major deliverable for this meeting was the specification. The actual specification developed is not included here because it failed to properly demonstrate functional specifications. During the semester, it was dropped from configuration control and not used by the design team at all. A better example is the specification in Appendix S, prepared by a similar class at The Wichita State University for the Gemini simulator project. It is a much more refined statement of requirements and functional specifications than the corresponding Ares document.

Assignments for the next two-week block were:

Roles for Class Project: Due 14 October 1986

Principal Architect: Lead preliminary design effort, monitor the partitioning of design tasks.

Project Administrator: Prepare new cost projections and analysis of two-week block. Continue to control computer budgets.

Configuration Manager: Conduct your business as per CM plan.

Quality Assurance Manager: Conduct your business as per QA plan.

Test and Evaluation Engineer: Prepare preliminary test plans, based on the requirements and specification documents, consisting of the overall strategy to be used, procedures for implementors to follow when doing unit tests, and organization of component tests. There should be a separate report from each T&E group.

Designer: Prepare preliminary design document, including all aspects of the external design and the architectural portion of the internal design. Electronic or hard-copy versions are to be delivered to me by 1300, 13 October. They should include screen layout/file format for all I/O, consistent variable identifiers, structure charts, and module interface tables. Data dictionary items should be upgraded.

Implementor: Write a program to act as a line editor. The program should be able to add, delete, and insert lines into a text file interactively. The lines should be stored using an array implementation of a linked list. Deliverable is the source file for this assignment, which should be readable without other documentation.

Documentation Specialist: Begin writing user manual based on specification document.

Verification and Validation Engineer: Conduct an inspection of the specification document to see if it is a valid form of the requirements. Develop test plans based on the specification documents. *This is the first instance where the IV&V team acted their roles outside of their own group. Verification inspections done at each milestone greatly contributed to the quality of the development documents.*

4.3.11. Class #11: Exam One, 2 October 1986

This is the first exam, designed to test the students on the objectives accomplished thus far.

15-413, Fall, 1986, Exam One

Name_____

Directions: Answer the following on a separate sheet. Use complete sentences. *As a major part of a software engineer's job is document production, I insisted on the proper use of English throughout the course. As you may surmise, there was a tremendous backlash when some students' answers were graded lower due to incomprehensibility.*

1. Describe the steps of the "waterfall" software development life cycle, identifying the intermediate products developed at each step. (32 pts.)

2. Compare and contrast two models of software development organizations in terms of their advantages/disadvantages. (10 pts.)

3. List and define three methods of requirements determination. (9 pts.)

4. Define the function of quality assurance and list two activities of QA personnel.
(9 pts.)

5. Define the function of configuration management and list two activities of CM personnel. (9 pts.)

6. Why is the "mythical man-month" mythical? Why does software development not fit the man-month model? (5 pts.)

7. What is the key metric used in nearly all cost and size estimating tools? (3 pts.)

8. Define a "walkthrough." (3 pts.)

9. Explain the difference between a requirement and a specification. (6 pts.)

10. Choose two design methods and compare them in terms of their applications domain and key characteristics. (10 pts.)

11. Specify two criteria for modularity. (4 pts.)

4.3.12. Class #12: Preliminary Design Review, 14 October 1986

Objectives:

- Conduct a review of the preliminary design.

Activities:

1. COLLECT ASSIGNMENTS. *Aside from individual assignments, these included the preliminary design document and the test plans of the Pascal and Ada teams, as well as an outline of the independent validation tests.*

2. QA leads a walkthrough of the preliminary design document.

Assignment:

Read Fairley, pp. 152-161 and 181-188.

The design groups made a critical decision during this time, which caused problems later. They decided to combine forces on the preliminary design, since it dealt with input and output only, in an attempt to simplify matters. One advantage to this approach was that only one user manual had to be produced. The difficulty was that it increased the number of people trying to settle design decisions from three to six. The conflict and communication index rose exponentially. The actual preliminary design was later incorporated into the detailed design, Appendix T.

The test plans produced by the Pascal testers (Appendix U) and the Ada testers (Appendix V) represent their thoughts about approaching the unit and integration testing of the different versions of the product. It is important to keep in mind that these testers are representing the development organization, whereas the work of the independent verification and validation team represent the "company." Therefore, much testing was done twice, but from different perspectives.

Assignments for the next two-week block were:

Roles for Class Project: Due 28 October 1986

Principal Architect: Lead final design effort.

Project Administrator: Prepare new cost projections and analysis of two-week block. Continue to control computer budgets.

Configuration Manager: Conduct your business as per CM plan. Submit hard copy of CR/DRs to date. *I needed to start tracking the change traffic.*

Quality Assurance Manager: Conduct your business as per QA plan. Submit summary of your two-week activities. *This is a way of getting an intermediate audit of QA activities.*

Test and Evaluation Engineer: Prepare test plans for external design. Conduct peer review of external design implementation as it is written.

Designer: Prepare final design document detailing processing logic.

Implementor: Implement external design. *This turned out to be a source of difficulty. The implementors paid no attention to the interfaces defined by the designers, causing severe problems integrating the later parts of the implementation.*

Documentation Specialist: Complete draft of user manual.

Verification and Validation Engineers: Conduct an inspection of the preliminary design to see if it properly implements the specification. Generate DRs as appropriate. Prepare test plans for external design. *This is a continuation of the incremental development of a comprehensive set of validation tests and tools. It was at this point that the IV&V team began developing automated test-generation tools to supplement the manually developed tests. These tests and tools are documented in the items located in Appendix W. The core of these documents was begun at this point in the class, and then added to as the semester progressed.*

4.3.13. Class #13: Design Tools, 16 October 1986

Objectives:

1. Discuss how changes identified at the design stage get incorporated into the requirements.

2. Present and discuss the applications of detailed design techniques.

Activities:

1. Discuss the following problems:

   - What happens at the design stage when a change is obvious? (You request the change. If it is made without approval, it is discovered and flagged via inspections. But the latter is NOT the ideal method.)

   - What happens at the implementation stage when changes are needed? (Same process.)

   This little exercise was in reaction to the implementors' "winging" changes in the preliminary design, which was already a configuration item.

2. Review the following detailed design methods, discussing pros, cons, and applications:

   - Data flow diagrams — unlike flowcharts, they do not indicate logic

   - Structure charts

   - HIPO diagrams: used for top-down design

   - Procedure templates: we used these already

   - Pseudocode: adds additional thought to logic, also syntax free

   - Structured flowcharts: "p-code in boxes"

   - Structured English

   - Decision tables

Assignment:

PARTITION THE WORK. *This reminder was included because the design group was trying to function in a consensus management style that was slowing down the development. I tried to get them to split up responsibilities for the detailed design.*

Read Fairley, pp. 192-224, 252-263.

4.3.14. Class #14: Implementation, 21 October 1986

Objectives:

1. Present the concept of structured programming.

2. Discuss the concept of exception handling.

3. Review language issues relating to implementing a design.]

4. *This class takes material learned in prerequisite programming courses and puts it into the context of software engineering.*

Activities:

1. Review content of Bohm and Jacopini paper on the three programming structures. *This is one of the few areas where some science directly underlies something useful in software engineering.* Emphasize single entry/single exit, linear control flow, three primary structures (sequence, selection, iteration), and the expansions of these (CASE, REPEAT...UNTIL).

2. Discuss GOTO: generally harmful except usable to create structure where none is present. For example, in 68000 assembler:

```
STRTWT: BTST    #1, CONTROL  * if ready for output then start
        BEQ     STRTWT       * else wait
```

Or BASIC:

```
10              READ X
20              IF X >= 0                        THEN 40
30                                               GOTO 70
40 REM          THEN
50              PRINT X, 'IS GREATER THAN 0'
60                                               GOTO 90
70 REM          ELSE
80              PRINT X, 'IS LESS THAN 0'
90 REM          ENDIF
```

3. A WHILE example in assembly language:

```
        MOVE.W  #MAXLOC,D3      * while i<=MAXLOC
WHILE:  MOVE.W  A2,D2
        CMP     D2,D3
        BEQ     ENDWHL
        ADD     (A2),D1         *  checksum <-- checksum+^i
        ADDQ.1  #2,A2           *  i <-- i+1
        BRA     WHILE           * endwhile
```

4. Present another style note for assembly language: creating macros that are roughly equal to high level language constructs.

5. Discuss use of user-defined types for clarity.

6. Examine procedures of less than 5 or more than 25 statements to see if they can be eliminated or simplified (the overhead in calling a five-line procedure exceeds that of repeating it in the source where needed).

7. Discuss standards versus guidelines and the stifling of creativity argument. *This deals with the argument that standards shut down creativity — they don't: you have to be pretty creative to meet them!*

---

8. Discuss exceptions: termination model versus resumptive model and the pros and cons.

9. Discuss concurrency: shared variables, test and set, asynchronous versus synchronous operation, Ada tasks, and the *accept, delay,* and *select* statements.

Assignment:

Work on design. Bring unresolved discrepancy and change reports to class.

4.3.15. Class #15: Configuration Control Board Meeting, 23 October 1986

Objectives:

- Demonstrate a configuration control board meeting.

Activities:

1. Conduct a CCB meeting to close out unresolved items before the release of the detailed design. *By this time the students will have held a couple of CCB meetings. It is critical to order the students not on the CCB to keep their mouths shut, or*

2. *you will have a riot on your hands.*

Assignment:

Finish the current set of bi-weekly project assignments. *Note that there is some more time left for detailed design work.*

4.3.16. Class #16: Critical Design Review, 28 October 1986

Objectives:

- Demonstrate a critical design review.

Activities:

- Conduct a critical design review. *Again, QA handled this, based on their plan.*

Assignment:

Read Fairley, pp. 283-296.

Assignments for the next two-week block were:

Roles for Class Project: Due 11 November 1986

Principal Architect: Stay out of the way. *I was trying to get the designers to live with their design.*

Project Administrator: Prepare new cost projections and analysis of two-week block. Continue to control computer budgets.

Configuration Manager: Conduct your business as per CM plan. Submit hard copy of CR/DRs to date. *By this time I was signing off on about 30 CR/DRs per two-week block.*

Quality Assurance Manager: Certify tests of external design and file summary report.

Test and Evaluation Engineer: Prepare test plans for detailed design. Execute test plans for external design. Conduct peer inspections as requested.

Designer: Answer implementor's questions; help maintain integrity of design document.

Implementor: Code everything. *This was a general go-ahead for all implementors.*

Documentation Specialist: Work on next draft of user manual by sending it to designers for technical review.

Verification and Validation Engineers: Conduct an inspection of the design to see if it properly implements the preliminary design. Generate DRs as appropriate. Prepare test plans for design. Execute test plans for external design.

4.3.17. Class #17: Software Testing and Integration, 30 October 1986

Objectives:

1. Introduce testing, verification, and validation.

2. Present the concepts of unit testing, integration, and acceptance testing.

3. Discuss debugging techniques.

Activities:

1. Differentiate between testing, verification, and validation.

2. Discuss unit testing:

    - Who does it
    - Syntax vs logic errors
    - Functional tests (derived from requirements)
    - Performance tests
    - Stress tests
    - Structure tests
    - White-box vs black-box testing

3. Discuss coverage criteria:

    - Statement coverage
    - Branch coverage
    - Logical path coverage (refer to Ada example on F286 that missed a path)
    - How much coverage attained/needed?

4. Discuss debugging:

    - Inductive (collect facts, create and test hypotheses)
    - Deductive (create hypotheses, evaluate each)
    - Backtracking (error isolation)

5. Discuss execution histories vs interactive debugging.

6. Discuss system testing:

    - Integration testing vs. acceptance testing (who does this?)
    - Top-down, bottom-up, big-bang, sandwich

7. Discuss cost of compiling big systems.

Assignment:

Read Fairley, pp. 267-283.

Notes re: project:

    1. CCB meets bi-weekly (principle of responsiveness).

    2. Must now coordinate code, test, IV&V, and system test.

    3. Draw flow chart on the board.

The purpose of this set of reminders was to reinforce for everyone the interaction between the various parts of the team. Coding, unit testing, integration testing, independent verification and validation, and quality assurance activities were all going on at once. In addition, the documentation team was pushing the technical people for reviews of drafts, and everyone was mad at the CCB. By this time, the CCB was backlogged so badly that I detected some cheating on the part of the coders. Things were being hacked rather than properly written and tested. I personally chewed out the two implementors involved and required the CCB to increase its meeting frequency to twice weekly. This helped significantly.

At about this time, I decided to implement 60 percent of the full design. The Clock, Crew, Activities, Input, and Output modules provided a working system. The Resources and Equipment modules were stubbed. In this fashion we could thoroughly test the 60 percent, as can be seen by the final reports of the IV&V team in Appendix W. I have discovered by experience that it is not fair to hold out for the whole thing, as that often forces too many all-nighters on the part of the testers.

4.3.18. Class #18: Verification and Validation I, 4 November 1986

Objectives:

1. Differentiate between life-cycle and formal verification.

2. Discuss QA activities.

3. Describe walkthrough and inspection techniques.

4. Evaluate static analysis and symbolic execution.

Activities:

1. Ask: What is the difference between life-cycle verification and formal verification?

2. Discuss verification (Are we building the product right?) and validation (Are we building the right product? What relates to what part of the cycle?).

3. Point out that V&V are pervasive and are not just for the end of the project. Ask: How has V&V functioned in this project at each step?

4. Note that QA and CM activities are mixed. Explain differing models.

5. Discuss audits: In-Process, Functional, Physical. What do these mean?

6. Review walkthroughs: process (implementor, moderator, qa); the goal is to discover, not fix errors; the key is a proper non-threatening atmosphere; two hour limit.

7. Discuss inspections: process (implementor, moderator, design, test); Fagan's results.[11]

8. Ask: What is static analysis? Symbolic execution? Utility?

Assignment:

Read Fairley, pp. 296-306.

---

[11]M. E. Fagan, "Design and Code Inspections To Reduce Errors in Program Development," in the *IBM Systems Journal*, Vol. 15, No. 3 (July 1976).

4.3.19. Class #19: Verification and Validation II, 6 November 1986

Objectives:

    1. Present concepts of formal verification.

    2. Discuss cons and pros of using formal verification and when it is most effective.

Activities:

    1. Discuss:

- Input/output assertions.
- Weakest preconditions.
- Cons: labor intensive, most literature dated, difficult to automate.
- Pros: good for essential, small algorithms such as security modules.

I found that wearing a tuxedo to the formal verification lecture distracts the students sufficiently so that you can sneak some content in on them.

Assignment:

Finish bi-weekly project assignments.

4.3.20. Class #20: Code Inspection, 11 November 1986

Objectives:

- Demonstrate a code inspection.

Activities:

- Conduct a code inspection using Fagan's techniques. *I used a segment of the code from the input module as an example.*

Needless to say, all the code was not done and unit tested, a situation that persisted for another week. The pressure placed on the coders by the other members of the class in limbo was tremendous, much worse than I could ever apply.

Assignment:

Work on project.

The final set of bi-weekly assignments follows:

Roles for Class Project: Due 2 December 1986

Principal Architect: Assist in attaining document consistency; assist in integration.

Project Administrator: Continue usual activities and create a lines-of-code/lines-of-comments table for each coded object of the detailed design. Comments embedded in a line of code do not count as comments. Null lines are to be ignored.

Configuration Manager: Conduct your business as per CM plan. Submit hard copy of CR/DRs to date.

Quality Assurance Manager: Certify that test plans have been executed. Certify that documentation meets the standards.

Test and Evaluation Engineer: Assist in integration. Execute test plans.

Designer: Help maintain integrity of the design document. Assist in integration.

Implementor: Assist in integration.

Documentation Specialist: Distribute manual for technical review. Incorporate changes/suggestions.

Verification and Validation Engineers: Conduct tests on product components and integrated product. Conduct code inspections of uninspected objects.

4.3.21. Class #21: Post-Development Software Support, 13 November 1986

Objectives:

1. Discuss the role of maintenance in software evolution.

2. Present methods of easing the maintenance activity.

Activities:

I had the opportunity to bring in a real live maintenance programmer, and I asked him the following questions:

1. What is the nature of your job? The type of product?

2. What is a typical task you do?

3. What documentation do you have available?

4. What hardware/software tools do you have?

5. What methods do you use when making a change?

6. How do you test changes?

7. How do you use test sites?

8. What else would you like to have to do your job better?

If I had presented this material myself, I would have stressed the sorts of tools and documents needed to do effective maintenance and discussed designing with maintenance in mind and knowing when to trash a product rather than fix it. As it was, these issues came out during the presentation.

Assignment:

Work on project.

4.3.22. Class #22: User Documentation, 25 November 1986

Objectives:

1. Discuss the key elements of good user documentation.

2. Discuss interaction between software engineers and technical writers.

Activities: *I always have a guest for this class meeting. If the students work for a small company, they may have to write user documentation. If they work for a company large enough to have technical writers, they have to know what to expect when dealing with them. I ask the guests to approach both these topics.*

Assignment:

Work on the project.

4.3.23. Class #23: Flight Readiness Review, 2 December 1986

Objectives:

- Demonstrate a release review of a product.

Activities:

1. Review the current status of the coding and testing of the individual modules.
2. Review the integration test results.
3. Determine what items will be waived prior to release.
4. Determine what work needs to be done in the remaining week.

This is obviously where the final decisions are made as to what will go out the door. We had the 60 percent coded and unit tested. Most of the integration was done. The actual status can be determined from reading Appendix W carefully.

Assignment:

Study for Exam Two.

Prepare for the final review. *See Class 25 below.*

4.3.24. Class #24: Exam Two, 4 December 1986

15-413, Fall, 1986, Exam Two

Name_____

Directions: Answer the following on a separate sheet. Use complete sentences.

1. What is the key difference between a data flow diagram and a flowchart? (5 points)

2. What is an important advantage of using pseudocode as a design representation? (5 points)

3. What is the scientific basis for the concept of structured programming? (15 points)

4. Differentiate between verification and validation. (10 points)

5. Differentiate between a walkthrough and an inspection in terms of goals, techniques, and effectiveness in error identification. (10 points)

6. Briefly define the following:

- functional tests
- performance tests
- stress tests
- black-box testing
- white-box testing
- coverage criteria
- integration tests
- acceptance tests (20 points)

7. Define three integration strategies and compare them in terms of techniques, advantages, and disadvantages. (15 points)

8. What are the advantages and disadvantages of formal verification? (10 points)

9. What is the best way to organize a user document? (10 points)

4.3.25. Class #25: Final Evaluation, 9 December 1986

To wrap up the semester, and to provide some insight into the course for those students taking it in the future and for faculty members, we presented a technical seminar at the Software Engineering Institute. I have held a seminar at the end of each version of this course and found it a positive experience for the students. They finally realize how much they have done and how much respect they have earned.

Agenda for 15-413 Software Engineering Final Evaluation

Held at the Software Engineering Institute, 9 December 1986

Room 201 at 1330

1330-1340	Introduction and Ground Rule<s	Jim Tomayko
1340-1350	Description of the Product	Chris Fedor
1350-1400	Project Accounting and Resource Management	Eric Borm, Steve Gale
1400-1410	Configuration Management Activities Walt Smith, Jon Lange	
1410-1420	Quality Assurance Activities	Blackwell & Cooper
1420-1445	Independent Verification and Validation Activities:	
	Overview Rich Wonsonegoro	
	Inspections	James Barton
	Ada V&V Frits Habermann	
	Pascal V&V	Brad Willitts
1445-1455	User Documentation	Dan Bernstein
1455-1510	Break	
1510-1540	Design Method	Phil Aspenwall
	Pascal Design Differences	Ruth Matsumura
1540-1550	Ada Implementation and Integration Doug Bunting	
		Sanjay Agrawal
1550-1600	Pascal Implementation and IntegrationDave Johnson	
		Todd Ihrig
1600-1630	Critique by SEI Staff — *a very useful segment*	

5. Epilogue

It is my hope that this case study gives potential software engineering instructors sufficient background and encouragement to offer project-intensive introductions to software engineering. This course is among the most rewarding of any in the computer science curriculum in terms of the joy of working closely with students and actually building something of practical use.

Seven of the original class members — Mark Abramowitz, Phil Aspenwall, Dan Bernstein, Eric Borm, Dave Johnson, Magnus Kempe, and Marc Scheurer — signed up for a three-credit project class the next semester and finished both the Pascal and Ada coding and testing. The release version of the software was presented to the public at the *Case for Mars III* Conference in Boulder, Colorado, July 1987.

References

[1]   James E. Tomayko.
        Software Configuration Management, SEI-CM-4-1.3.
        September, 1987.

[2]   James E. Tomayko.
        Support Materials for Software Configuration Management, SEI-SM-4-1.0).
        September, 1987.

Appendix A: Appendices A — Z Order Form

Order Form for Appendices

To order the appendices, please return this form to:

> The Software Engineering Institute
> Carnegie Mellon University
> Pittsburgh, PA 15213
> Attn: Allison Brunvand

❑   Teaching a Project-Intensive Introduction to Software Engineering $55.00

❑   Academic Affiliates Special Price (1st copy only) $20.00

     Total .... $____

All orders must be prepaid. Please indicate the method of payment:

❑   Check

❑   Purchase Order

❑   Money Order

Ship the appendices to:

Name ....

Title ....

Address ....

....

City .... State .... Zip Code ....

Table of Contents