Technical Report

CMU/SEI-91-TR-10
ESD-91-TR-10

Models for Undergraduate Project Courses in Software Engineering

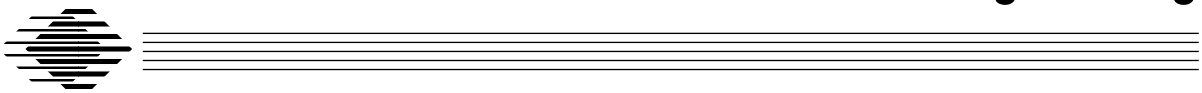
Mary Shaw
James E. Tomayko

August 1991

Technical Report

CMU/SEI-91-TR-10
ESD-91-TR-10
August 1991

Models for Undergraduate Project Courses in Software Engineering



Mary Shaw

School of Computer Science
and Software Engineering Institute

James E. Tomayko

School of Computer Science
and Software Engineering Institute

This report will also appear as Carnegie Mellon University
School of Computer Science Technical Report No. CMU-CS-91-174.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

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.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc. / 800 Vinial Street / Pittsburgh, PA 15212. Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page at http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service / U.S. Department of Commerce / Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides acess to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential con tractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: 1-800-225-3842 or 703-767-8222.

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

Models for Undergraduate Project Courses in Software Engineering

Abstract: The software engineering course provides undergraduates with an opportunity to learn something about real-world software development. Since software engineering is far from being a mature engineering discipline, it is not possible to define a completely satisfactory syllabus. Content with a sound basis is in short supply, and the material most often taught is at high risk of becoming obsolete within a few years.

Undergraduate software engineering courses are now offered in more than a hundred universities. Although three textbooks dominate the market, there is not yet consensus on the scope and form of the course. The two major decisions an instructor faces are the balance between technical and management topics and the relation between the lecture and project components. We discuss these two decisions, with support from sample syllabi and survey data on course offerings in the United States and Canada. We also offer some advice on the management of a project-oriented course.

1. Introduction

For most undergraduate students, an upper-level software engineering course provides the best opportunity to learn about "real-world" software development: group projects, large-scale design, integration of software with larger systems, applying theory in practical settings, understanding clients' requirements, models of the development life cycle, configuration management, quality assurance, maintenance, and so on. In many universities, it is our last and best chance to show students that developing a real software system is not at all the same as writing a programming assignment that will be graded and thrown away. The course is most often taken by students about to enter the work force as programmers; our experience is that corporate recruiters are very enthusiastic about the things students learn in these courses.

Versions of the software engineering project course have been offered for nearly twenty years, but despite that extended history the course is far from standard in either content or format. A great deal of material is available—much more than can be covered in a quarter or a semester—so the course design problem is one of selecting the subset to present and the viewpoint from which to present it.

Any curriculum design task is in part a resource allocation task. The problem is determining the amount of content that can be presented, constrained by the scarce resources of class time, student energy, and testing attention, among other factors. The evaluation criterion for selecting content is benefit to the student through his or her career, and we often describe course content having this durable value as "fundamental." Along with fundamental

material, software engineering includes techniques (often management strategies) that are currently useful but are likely to be superseded as fundamental material emerges. As a result, the challenge to the software engineering instructor is to find meaty, substantive material that will continue to be significant to the student over many years.

An instructor faces two major decisions in selecting a strategy for the software engineering course. The first is what balance to strike between technical issues of software design and development on the one hand and project management topics on the other. Section 2 discusses the considerations that affect this decision. The instructor's second decision is how to allocate class effort between project work and lectures—and how to coordinate the two. Section 3 describes several course models that make different tradeoffs on this dimension, and Section 4 discusses the *kinds* of projects used in undergraduate courses.

In doing this analysis we had the benefit of the Software Engineering Institute's survey of software engineering courses [84] and personal contact with some two dozen instructors. Section 5 and the appendix present some objective data from the spring 1990 survey, including the position of the courses in the curriculum and the textbooks in use.

To put flesh on the bones of this overview, Sections 6 through 10 present the plans of several versions of the course, including syllabi of actual offerings, outlines of textbooks, and curriculum designs.

Finally, no matter which decision the instructor makes on the two major questions of content and organization, any course with a significant group-project component presents several special problems. Section 11 notes some of these and suggests ways to deal with them.

Software engineering education has regularly been singled out for special attention. An early workshop was held in 1976 [129]. An annual series of workshops on software engineering education has been sponsored since 1986 by the Software Engineering Institute [57, 48, 50, 56, 39]. Papers on particular topics also appear regularly in the ACM *SIGCSE Bulletin* and in special issues of other professional journals.

2. Content Balance: Technical versus Management

Traditionally, engineering is defined as "creating cost-effective solutions to practical problems by applying scientific knowledge to building things in the services of mankind" [114]. Against that standard, the phrase "software engineering" is a statement of aspiration, not a description of accomplishment. Nevertheless, we face a crying need to get better control over the software development task and to do so in the short term.

Engineering disciplines have historically emerged from ad hoc practice in two stages. They are rooted in the exploitation of a technology by individual craftsmen. As the technology becomes more significant, management and uniform production techniques are developed to support a stable market. Problems with production in this commercial setting stimulate the development of a supporting science. As the science matures, it merges with established practice to enable a professional engineering practice rooted in the science [114]. Figure 1 depicts this evolution.

Figure 1: Evolution of an Engineering Discipline

Certainly the software development task is appropriately an engineering problem: it involves "creating cost-effective solutions to practical problems." What's currently lacking is widespread routine application of scientific knowledge to a wide range of practical design tasks. Some science is mature, most notably algorithms, data structures, and compiler construction theory; however, the current demands are far ahead of the scientific base. In other words, the field of software engineering is still immature; but the situation is improving, albeit more slowly than need requires.

In the absence of a mature scientific base, developers of large software systems resort to establishing routine practices which, even if ad hoc, will nevertheless guide programmers to relatively predictable outcomes. As a result, the "software engineering" label is being ap-

plied to a set of management topics such as project planning, cost/schedule estimation, and team organization.

These topics may represent best current practice, but:

- The pipeline of scientific results with practical utility is filling, particularly in specific application areas.
- Course energy will have the highest payoff if invested in the most durable content.
- Use of the "engineering" label for what are essentially management topics may preempt its use for real engineering techniques—and may even divert attention from the cultivation of those techniques.

Here lies the source of the instructor's dilemma: whether to emphasize the mature science or the current practice. The former is less applicable at present but has been refined enough to retain its value for years to come. The latter represents useful skills that may become obsolete—and may also be inefficient in course time per unit of content. The decision is complicated by the need to present material that is required simply to enable students to carry out a sizable group project.[1] For concreteness, we list here typical technical and management topics. It is difficult to fully separate the two, for a given subject can sometimes be approached with either emphasis. For example, "configuration management" has both technical content (formal methods of system description) and management aspects (composition and activities of a configuration control board). This dichotomy exists for several other areas within the scope of software engineering: life cycle models, quality assurance, maintenance topics, etc.

Technical engineering topics

- Abstraction and specification, and their proper use in design
- Formal methods of requirements specification and of verification
- Large-scale data, including scientific, commercial database, and AI issues
- Security and reliability
- Distributed systems, including synchronization/atomicity and network communication

[1] An additional argument for including management topics arises from employers' needs: it has been our experience that students with a solid background in software engineering currently advance quite rapidly after graduation to team leadership positions in which they need to know those skills. In one instance, a Carnegie Mellon graduate found himself leading a small independent research and development team in a major aerospace company within four months of leaving school. He reported that his supervisors chose him for the lead because he had a better understanding of the overall software development process than more experienced computer science graduates who had neither studied software engineering nor, as yet, gained an appreciation for the "big picture."

- Real-time systems
- Concepts of prototyping
- Development of metrics for measuring quality factors
- System description languages
- Techniques for software reuse
- Automated development environments and their characteristics
- Configuration management
- Structuring software for maintenance

Management topics

- Life cycle models as a framework for development planning
- Requirements analysis
- Estimation and tracking
- Nature and use of development standards
- Configuration management
- Development team organization, structured design methods (e.g., Yourdon, Jackson, SADT)
- Tracking progress through internal documents
- The use of quality assurance
- Reviews and walkthroughs
- Maintenance

In general, the course plans that we have reviewed show a strong management emphasis both in the topics discussed in class and in the project work. The most often-used textbooks, whose outlines are presented in Section 6, certainly show that emphasis. For comparison, an example of a course based on a technical theme, in this case abstraction, is given in Section 7. We can suggest several reasons for the predominance of courses with a management emphasis:

- The most popular texts spend a considerable percentage of their content dis-

cussing life cycle models, management issues relating to organizing software teams, quality assurance and configuration management from the standpoint of their organizational impact rather than their technical activity, etc.; this textual emphasis creeps into the syllabi of inexperienced instructors.

- Instructors with industrial experience recognize that the chief failings of software projects in the commercial world are not technical but are problems such as poor effort and resource estimates, poor communication, and lack of configuration management and quality assurance.

- A certain amount of management material is needed to keep a group project running well. Since this material is qualitatively different from the topics students are accustomed to seeing in computer science courses, it is likely to be more time-consuming to teach.

- Most of the students plan to go directly into industry after graduation, and the software engineering course is the only place in the undergraduate computer science curriculum where these topics can be adequately taught.

Alas, these reasons put vocational considerations ahead of the traditional criteria of curriculum design.

Until very recently, the technical issues associated with software engineering fit quite well within the course alongside the management issues—there was so little to discuss that there was plenty of class time for both. The students usually enter the course with a solid grounding in program construction, so the challenge appeared to be meshing the management techniques with the software development techniques. Software engineering as a technical discipline has hardly stood still: formal methods, new design paradigms, and other technical topics are squeezing the syllabus from one end, while process maturity models, post-development management issues, and other management topics are squeezing it from the other. There is now a very real need to make decisions on content that are different from the decisions early in the last decade.

Various decisions about the balance of technical and management topics can be supported for different institutional settings and variations on the basic objective of learning about "real-world" software. But to justify the title "software engineering," the course must convey to students the engineering point of view: that engineering of software involves generating alternative designs and selecting—for good reasons—the ones that best resolve conflicting constraints and requirements of the ultimate customer.

3. Models of the Undergraduate Software Engineering Course

Beginning software engineering courses have been taught with content that crosses a range from no project work at all to highly intensive project work. Figure 2 is an illustration of the spectrum of courses. The models described in this section are drawn from extensive discussions with instructors of these courses over seven years, augmented by the literature.

Figure 2: Models of the One-Semester Course

An underlying assumption of these models is that students enter the course with the technical background of program construction gained in the prerequisite courses, and the purpose of the software engineering course is to demonstrate how this technical material is applied in the context of large-scale software development. Students are expected to have

knowledge and skills in the application of high-level programming languages, especially those that emphasize structure and abstraction, knowledge of machine organization (perhaps demonstrated by facility with assembly language), and grounding in the fundamental principles and algorithms of computer science. Further, students should be good writers and should have senior standing. This last requirement is to ensure that the students have had some upper-division courses such as theory, compiler design, or survey of programming languages, and that they have some maturity in both their academic and personal lives. Jon Bentley once observed that a useful prerequisite for the study of software engineering is that the student "should have been married at least once." This is not strictly enforced(!), but the students should have some ability to communicate and work together somewhat harmoniously.

3.1. The "Software Engineering as Artifact" Model

The leftmost model illustrates the subject taught almost entirely by lecture, with some interaction among the instructor and students, mostly relating to questions and difficult points. There are two advantages of 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. Because 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. Due to the nature of these courses (and for reasons discussed in Section 2), instructors often follow the plan described in Section 6, "Life Cycle Emphasis."

3.2. The "Topical Approach" Model

Although the "topical approach" model is another all-talk, no-action model, it has the advantage of in-depth exploration of some aspects of the subject. Usually used at the graduate level, the lecture part of the course is roughly the same as in the previous model, but it is supplemented by weekly presentations by the students. Each student is assigned a topic (such as object-oriented design, automated specification tools, and verification of real-time software), reads two or three papers on it, and conducts a seminar for the other students. Here a succinctly written text such as one by Fairley [47] or Sommerville [118] can be used to back up the main lectures, while the students provide their own reading material for the discussion sessions. Alternatively, essay collections such as Brooks [24] or Mills [89] can feed the discussions. Yourdon's publishing arm has also produced a pair of collections of outstanding papers in software engineering [134, 135], though some are becoming dated. Again, even though there are additional readings, these courses often follow the plan in Section 6.

3.3. The "Small Group Project" Model

The "small group project" model includes a project as part of the course, for the positive reasons discussed above. Currently the most common model of the software engineering course, it makes a fairly even division between project work and class work. Typically, students are divided into teams of three to five members each. The projects chosen are often familiar to the students and can be completed in a single term. However, the limited course effort dedicated to the project necessarily limits its size and scope.

In Kant's presentation of this model [69], she suggests some projects: a string-handling package for standard Pascal, a reservation system for the computer center's terminal room, and an interactive text editor, among others. This model provides the students with some of the experience needed to apply the software engineering concepts discussed in class. However, the model is deficient in its ability to give students experience with the critical difficulties inherent in programming-in-the-large.

The use of a small project, small teams and, often, fictitious customers may simply extend the programming-in-the-small experience normally gained in computer science courses. However, adequate attention to configuration management and quality assurance can bring in larger project issues. One way of taking advantage of the manageability of small teams while still teaching something about programming-in-the-large is to use the course organization described in Section 10, "Even Balance of Management, Engineering, and Tools." Another reason that small teams are used is that the start of the project is delayed until about the middle of the term so that some prerequisite technical material can be taught [63]. This often happens when instructors use the course plan described in Section 7, "Abstraction Emphasis."

3.4. The "Large Project Team" Model

The "large group project" 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. Typically, 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, how is it possible to manage a real-life corporate development team of 15 to 30 software engineers? The authors have run introductory courses using this model at least 10 times, in general quite successfully.[2] Even with partial failure, the students learn more about real software engineering with this model than with any other. The key for the instructors 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.*

In this model, the students are organized in one large team with different roles such as those found in industrial software development environments. For example, some students may be designers, others quality assurance personnel, still others configuration managers. The students remain in these roles for the duration of the course, and learn about the activities associated with other roles through normal interaction with other team members while developing the software. In Sections 9 and 10, we describe implementations of this model.

3.5. The "Project Only" Model

The fifth column in Figure 2 represents the "project only" model: the entire course is a project, and the various topics of software engineering are learned and applied by immersion. This model is often found as the capstone experience of master's level software engineering programs, but several universities also allow undergraduates to take such a course, organizing the resulting teams with the graduate students in leadership positions and the undergraduates subordinate to them. The undergraduates learn by doing. The course plan in Section 8.2 is a version of this model, in which students pursue independent projects *after* taking a course that integrates project and lecture.

[2] Examples of other instructors successfully managing this sort of course are described in [94] and [110].

3.5.1. The "Separate Lecture and Lab" Model

In many ways, this model is a combination of the courses represented by columns one and five of Figure 2. Though nominally linked by related course numbering, software engineering lecture courses with separate labs have coordination reminiscent of freshman physics courses of twenty years ago. The amount of coordination is strongly related to whether or not the course instructor is also the lab instructor.

4. Project Styles

Independent of the organization that connects the lecture and project components of the course, several styles of projects are available. They differ primarily in the degree to which they stretch the student beyond traditional programming assignments and in the amount of interaction among the students taking the course. They also differ in emphasis and in the amount of instructor effort required.

In virtually all project courses, students are organized into teams. The major parameters in the project decision are:

- Does the project stand alone, or does it involve integrating hardware or software components from other sources?
- Is the project specified by the instructor, by the class (as the first task of the course), or by an external client? How detailed is this specification, and how much work must the students do to make the requirements precise?
- Do all teams complete the same assignment, or do the teams develop separate components and integrate them?
- Does each student play all roles in the project team, or is there specialization of individuals or teams?
- Is the emphasis on the technical development of software, on the organization and management of the team, on the interaction with the client (e.g., for requirements and delivery), or on something else?

Some of the common points in the space defined by these parameters are in the taxonomy of project styles described below.[3]

4.1. Toy Project

The class is divided into teams of 3-5 students; each team gets the same task, pre-specified by the instructor. In a variant, each team creates its own specification, perhaps within a given scope such as "create a computer game using this interface/library."

The advantages of this project style are that it teaches teamwork in a small group and it results in a large-ish program, but it rarely addresses requirements or integration. It is probably the simplest to manage from the instructor's viewpoint, and teaching assistants can be

[3]Two other studies of undergraduate software engineering courses also have taxonomies of project styles. Thayer and Endres [122] specify 11 examples of courses and projects, while Leventhal and Mynatt [78] derived only 3. These analyses make roughly the same distinctions as we have, but they do not clearly distinguish the separate concerns about project size (Section 3) and project style (this section).

effectively used as advisors to groups. Also, where there is a mix of graduate and undergraduate students in a course, the graduate students often lead the small teams. However, this style fails to transcend the development methods effective for most smaller programs and often ignores configuration management and other issues important to programming-in-the-large. Despite these disadvantages, Leventhal and Mynatt [78] report that 40% of all software engineering courses use this style.

4.2. Mix-and-Match Components

In this project style, the software product requires several components. Small teams develop each component (to precise specification), and several versions of the final software result from integrating different collections.

One instantiation of this style is Horning's "Software Hut" [64]. The project requires half to a third as many components as there are teams. In the first stage of the project, each team develops one component then "buys" components from other teams and "sells" its own, so each team gets a complete set. In the second stage, each team integrates components to produce a complete system. A third stage, requiring another round of sales and a modification task, may follow.

This project style teaches many of the same topics as the "large programming assignment" style; additionally, it gives the students some experience with rigidly defined interface specifications and integration techniques on a larger scale. The concept of software reuse can also be vividly demonstrated. Management of this style is nearly the same as for "toy project."

4.3. Project for an External Client

The student teams work on components of a product for an external client, integrate results, and then pass an acceptance test set by the client. This project style teaches teamwork, project organization, requirements extraction and representation, integration, and client relations. From the course management perspective, this style potentially requires much work from the instructor, especially on "customer relations." Whereas a simulated economy such as Software Hut is designed to be competitive, this style is intended to be cooperative—the success of all depends on successful delivery.

As described in Sections 9 and 10, either a group of small teams or a single large team can implement the project. Experience shows that the motivation of the students is significantly higher with an external client, most probably due to the prospect of seeing their software in actual use, perhaps by their peers.

4.4. Individual Projects

In this project style, each team has a separate project. Often the teams have clients outside the course; for example, the campus computing facility or departmental research projects. These clients are usually identified by the instructor. The course may be chaotic from a management standpoint, especially if it is essentially an independent-study course for groups of students rather than individuals. Otherwise, this style has many of the same features as the other small-team approaches. The laboratory course described in Section 8.2 is an example of an individual project course.

5. Survey of Software Engineering Courses

The Software Engineering Institute periodically conducts a survey of software engineering degree programs and courses [84]. The survey originally queried schools with graduate programs in computing and information systems; later it was extended to include schools with undergraduate offerings in those areas. The survey reported in April 1990 included 12 software engineering degree programs and courses at 165 schools in the United States and Canada. The 12 degree programs all lead to a master's degree and are not considered here.

The reporting from school to school was not uniform: some included software-related courses that do not appear to be at all like the course of interest here. After deleting a variety of courses in subjects such as data structures, artificial intelligence, security, and networking, we summarized the reported data on the remaining courses. After this pruning, the survey found 472 courses at the 165 institutions. Details are provided in the appendix.

A total of 126 schools reported 205 software engineering courses explicitly open to undergraduates. Of these, 144 were reported to be undergraduate courses and 61 were open to both graduates and undergraduates. Because the survey pool is biased in favor of schools with strong computing programs, it is reasonable to expect that the survey will find a greater percentage of undergraduate software engineering offerings than in the college and university population at large. Thus the availability of software engineering courses to undergraduates in only 75% of those schools suggests a substantially weaker market penetration overall. We speculate that the reasons might include resource constraints, shortage of faculty prepared to teach the course, or the fact that the course is listed only as an elective in "Curriculum '78" [2], the most recent recommendation from the professional organizations. Note, however, that more than half of the purely undergraduate courses are reported in the survey as required courses.

The most common course titles among the undergraduate courses are shown in Table 1. When we grouped these titles into rough categories, we found 87 software engineering courses (42%), 41 courses concerned with particular stages of the life cycle (20%), 28 project courses (14%), and assorted others.

Number	Course Name
30	Software Engineering (one term)
17	Software Design and Development, SW Design, SW Development, etc.
13	Systems Analysis and Design, Systems Analysis, Structured System Design, etc.
9	Introduction to Software Engineering
9	Software Engineering Project, Senior Project
8	Software Methodology, Programming Methodology
7	Software Engineering I and II (two terms)

Table 1: Most Common Titles for Undergraduate Software Engineering Courses

Leventhal and Mynatt [78] surveyed software engineering courses in 240 of the 820 undergraduate computer science programs listed in ACM's 1984 administrative directory. With a 25% response rate, they found 35 courses that match the portion of the SEI survey reported here. The most commonly used course title was "Software Engineering" or some variant (19 of 35); an additional 9 course titles referred to specific stages in the life cycle; the remaining 7 course titles were related to software development.

Well over a hundred textbooks are mentioned in the SEI survey; the vast majority are used in only one or two courses. Even when a textbook is assigned for a course, the course does not necessarily follow the text closely. However, of the information provided by the survey, the text and the course title are the best indicators of content. Table 2 lists the dozen or so most frequently used textbooks for all the courses surveyed, together with the number of times the textbook was cited for courses open to undergraduates. These account for about 60% of the courses. The appendix extends the list and compares graduate to undergraduate text selection.

CITES		TEXTBOOK
All	**Open to UGs**	
60	38	Pressman, *Software Engineering: A Practitioner's Approach* [104]
46	31	Sommerville, *Software Engineering* [118]
46	29	Fairley, *Software Engineering Concepts* [47]
32	20	manuals on languages and tools
32	5	selected readings
26	18	Booch, *Software Engineering with Ada* [21]
20	12	Brooks, *The Mythical Man-Month* [24]
15	1	Shooman, *Software Engineering: Design, Reliability, and Management* [116]
10	4	Yourdon, *Modern Structured Analysis* [136]
10	2	Conte, *Software Engineering Metrics and Models* [35]
9	5	Liskov, *Abstraction and Specification in Program Development* [80]
9	5	Myers, *The Art of Software Testing* [92]
9	5	Page-Jones, *The Practical Guide to Structured Systems Design* [98]
8	8	Lamb, *Software Engineering: Planning for Change* [77]

Table 2: Commonly Used Software Engineering Texts

6. Course Plan: Life Cycle Emphasis

College courses often use textbooks as resources rather than follow them strictly. However, the textbooks certainly show how their authors intended the courses to be structured. When an instructor is not fully confident of the subject matter, the text organization often prevails. In the case of the three dominant texts, this means that the waterfall life cycle model drives the course organization, with the emphasis heavily on the six stages of project planning, requirements analysis and specification, design, implementation, validation, and maintenance. Other topics are discussed, but often in an order that actually precludes good software development practice. For example, material on configuration management is relegated to the back of the texts, giving the impression that it either comes into play at that point in a software project or it is so unimportant as to be a candidate for exclusion if the other topics fill the semester.

The life cycle emphasis is clearest in Fairley's text [47], whose outline is given below in detail. The same emphasis is visible in Pressman [104] and Sommerville [118], for which we give only short outlines.

6.1. Fairley: *Software Engineering Concepts*

Objectives (from the Preface): Primary goals for this text are to acquaint students with the basic concepts and major issues of software engineering, to describe current tools and techniques, and to provide a basis for evaluating new developments. Many different techniques are presented to illustrate basic concepts, but no single technique receives special attention.... The layout of the text follows the traditional software life cycle: an introductory chapter is followed by chapters on planning, cost estimation, and requirements definition. These are followed by chapters on design, implementation issues, and modern programming languages. Chapters on quality assessment and software maintenance conclude the text. This layout should not be taken as an indication that the phased life cycle approach to software development is the only method presented in the text. The use of prototypes and successive versions is emphasized throughout the text and in the term project material.

Chapter Outline:

1. **Introduction to Software Engineering**
   - Some Definitions
   - Some Size Factors
   - Quality and Productivity Factors
   - Managerial Issues
   - Overview of the Text
2. **Planning a Software Project**

- Defining the Problem
- Developing a Solution Strategy
- Planning the Development Process
- Planning an Organizational Structure
- Other Planning Activities

3. **Software Cost Estimation**
   - Software Cost Factors
   - Software Cost Estimation Techniques
   - Staffing-Level Estimation
   - Estimating Software Maintenance Costs
4. **Software Requirements Definition**
   - The Software Requirements Specification
   - Formal Specification Techniques
   - Languages and Processors for Requirements Specification
5. **Software Design**
   - Fundamental Design Concepts
   - Modules and Modularization Criteria
   - Design Notations
   - Design Techniques
   - Detailed Design Considerations
   - Real-Time and Distributed System Design
   - Test Plans
   - Milestones, Walkthroughs, and Inspections
   - Design Guidelines
6. **Implementation Issues**

- Structured Coding Techniques
- Coding Style
- Standards and Guidelines
- Documentation Guidelines

7. **Modern Programming Language Features**
   - Type Checking
   - Separate Compilation
   - User-Defined Data Types
   - Data Abstraction
   - Scoping Rules
   - Exception Handling
   - Concurrency Mechanisms

8. **Verification and Validation Techniques**
    - Quality Assurance
    - Walkthroughs and Inspections
    - Static Analysis
    - Symbolic Execution
    - Unit Testing and Debugging
    - System Testing
    - Formal Verification
9. **Software Maintenance**
    - Enhancing Maintainability During Development
    - Managerial Aspects of Software Maintenance
    - Configuration Management
    - Source-Code Metrics
    - Other Maintenance Tools and Techniques
10. **Summary**
    - Planning and Cost Estimation
    - Software Requirements Definition
    - Software Design Concepts
    - Implementation Issues
    - Modern Language Features
    - Verification and Validation Techniques
    - Software Maintenance

6.2. Pressman: *Software Engineering: A Practitioner's Approach*

Chapter Outline:

1. Software and Software Engineering
2. Computer System Engineering
3. Software Project Planning
4. Requirements Analysis Fundamentals
5. Requirements Analysis Methods
6. Software Design Fundamentals
7. Data Flow-Oriented Design
8. Data Structure-Oriented Design
9. Object-Oriented Design
10. Real-Time Design
11. Programming Languages and Coding
12. Software Quality Assurance
13. Software Testing Techniques
14. Software Testing Strategies
15. Software Maintenance and Configuration Management

6.3. Sommerville: *Software Engineering*

Major Parts:

1. Software Specification
2. Software Design
3. Programming Practice, Techniques, and Environments
4. Software Validation

5. Software Management

7. Course Plan: Abstraction Emphasis

One example of a text that breaks the waterfall organizational paradigm is the one by Liskov and Guttag, which concentrates on abstraction as a tool and method for software development [80]. The authors introduce the technical basis for abstraction and a suitable language before presenting a model of the software development process. In the latter section, they are careful to provide explicit examples of design based on abstraction. In many ways this text is a good statement of the fact that there is some available technical underpinning for software engineering practice.

7.1. Liskov and Guttag: *Abstraction and Specification in Program Development*

Objectives (from the Preface): Good programming involves the systematic mastery of complexity. It is not an easy subject to teach. Students must be convinced that programming is not an arcane art, but an engineering discipline. There are useful principles that can and should be applied. In this book we attempt to shape the way students think about programming by presenting, justifying, and illustrating a cohesive doctrine. Our principal tenet is that abstraction and specification must be the linchpins of any effective approach to programming. We place particular emphasis on the use of data abstraction to produce highly modular programs.

Chapter Outline:

1. **Introduction**
   - Decomposition and Abstraction
   - Abstraction
2. **An Overview of CLU**
   - Program Structure
   - Integers, Booleans, and Arrays
   - Objects
   - Declarations and Assignment
   - Procedures
   - Expressions
   - Data Types
   - Input/Output

- Object-Oriented Programs

3. **Procedural Abstraction**
   - The Benefits of Abstraction
   - Specifications
   - Specifications of Procedural Abstractions
   - Implementing Procedures
   - More General Procedures
   - Designing Procedural Abstractions
4. **Data Abstraction**
   - Specifications for Data Abstractions
   - Implementing Data Abstractions
   - Using Data Abstractions
   - Implementing Polynomials
   - Aids to Understanding Implementations
   - Parameterized Data Abstractions
   - Lists
   - Ordered Lists
5. **Exceptions**
   - Specifications
   - The CLU Exception Mechanism
   - Using Exceptions in Programs
   - Design Issues
6. **Iteration Abstraction**
   - Specifications
   - CLU Iterators

- Examples
- Design Issues

7. **Using Pascal**
   - Procedure and Function Abstractions
   - Data Abstractions
   - Polymorphic Abstractions
   - Generators
   - A Full Example
8. **More on Specifications**
   - Specifications and Specificand Sets
   - Some Criteria for Specifications
   - Why Specifications?
9. **Testing and Debugging**
   - Testing
   - Unit and Integration Testing
   - Tools for Testing
   - Debugging
   - Defensive Programming
10. **Writing Formal Specifications**
    - An Introduction to Auxiliary Specifications
    - Interface Specifications of Procedural Abstractions
    - Interface Specifications of Data Abstractions
11. **A Quick Look at Program Verification**
    - Reasoning About Straight-Line Code
    - Reasoning About Programs With Multiple Paths

- Reasoning About Procedures
- Reasoning About Data Abstractions
- A Few Remarks About Formal Reasoning

12. **A Preamble to Program Design**
    - The Software Life Cycle
    - Requirements Analysis Overview
    - A Sample Problem
13. **Design**
    - An Overview of the Design Process
    - The Design Notebook
    - Problem Description
    - Getting Started on a Design
    - Discussion of the Method
    - Continuing the Design
    - The Doc Abstraction
    - The Line Abstraction
14. **Between Design and Implementation**
    - Evaluating a Design
    - Ordering the Program Development Process
    - The Relationship of Abstraction to Efficiency
15. **Using Other Languages**
    - The Stack Approach
    - Choosing an Approach

8. Course Plan: Technical Emphasis

In the early 1980s, Carnegie Mellon proposed a plan for a complete undergraduate curriculum; the curriculum was published in 1985 [113]. The plan included *both* a project-based software engineering course and a follow-on project-only course that provided extended experience. The outline syllabi for those two courses are included here. Note that the prerequisites include two nonstandard courses: one is about large-scale program organizations (architectures); the other combines elements of present comparative-language and compiler courses.

8.1. Software Engineering

Prerequisites: PROGRAM ORGANIZATIONS [313]
LANGUAGES, INTERFACES, AND THEIR PROCESSORS [320]

Description: The student studies the nature of the program development task when many people, many modules, many versions, or many years are involved in designing, developing, and maintaining the system. The issues are both technical (e.g., design, specification, version control) and administrative (e.g., cost estimation and elementary management). The course will consist primarily of working in small teams on the cooperative creation and modification of software systems.

Rationale: This course extends the advanced program structures course by broadening the scope of attention to large-scale systems. This yields a natural progression from individual elements (statements or data structures) in FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE I AND II [211/212] through module-sized elements in PROGRAM ORGANIZATIONS [313] to large systems. Analysis and evaluation techniques are included throughout, but the emphasis on estimation and overall efficiency is greatest here. In addition, issues of reliability, testing, and implementation and documentation of a substantial user interface will be addressed here.

Objectives: At the end of this course, a student will be able to:

- Understand the issues in large-scale software development
- Participate as a team member in such a development
- Write specifications for simple modules that will be combined with other modules
- Implement a program or module that satisfies such a specification

Ideas: This course will be the primary carrier of the following:

- Complexity of large-scale software and tools for dealing with it

It will reinforce or share responsibility for:

- Significance of tools for developing software

Topic Outline:

1. **Elementary management**

2. **Cost estimation (of routines and larger code units)**

3. **Multiple people, versions, years, modules, modifications**

4. **Advanced design and specification; decomposition into modules**

5. **Programming-in-the-large**

6. **Properties of systems**
   - Reliability
   - Generality
   - Efficiency
   - Complexity
   - Compatibility
   - Modularity
   - Sharing

7. **System design and development principles**
   - Design tradeoffs
   - Computer system reliability, speed, capacity, cost
   - Development methodologies and tools
   - Design automation
   - Program specification
   - Maintenance and release policy (test sites, etc.)
   - Rapid prototyping and partial evaluation
   - Protection and security
   - Resource allocation
   - System evaluation and development aids

8. **Modification, planning for modification**

9. **Making implementation meet specifications**

10. **Models and modeling**

- System modeling (version control)
- What models are and how to use/construct them
- Empirical versus analytic models
- Validation
- Specific models (at this level, introduction only)
  - Queuing-theoretic models for operating systems and hardware
  - Productivity and life-cycle models (esp. their limitations)

11. **Monitoring tools and techniques for improving efficiency**
12. **Human factors, user interfaces**

13. **Examples of systems**
    - Large software systems, some involving concurrency issues
    - Distributed systems
    - Compilers, operating systems
    - Batch versus time-sharing systems
    - File management
    - System accounting
    - The multiprogramming executive (MPX) operating system
    - Process control
14. **Current state of the art: APSEs, Gandalf, etc.**
15. **Software systems**
    - Systems and utility programs
    - System structure
    - Parallelism in operating systems
        - Mutual exclusion
        - Synchronization
16. **Programming style and techniques**
    - Table-driven schemes
17. **Management, societal, economic, and legal aspects**
    - Computing economics: acquisition and operation
    - Copyrights, licensing and patents, computer crime
18. **Documentation**
19. **Software systems**
    - Memory management
    - File systems
    - Directories

- Backup and recovery
- Permanent and transient data: caching, buffering, atomic transactions, stable storage
- Redundancy, encoding, encryption
- Database management systems (DBMS)

References:

B. W. Boehm, *Software Engineering Economics*.

F. Brooks, *The Mythical Man-Month*.

G. Myers, *Software Reliability: Principles and Practices*.

G. Myers, *Composite/Structured Design*.

M. Shooman, *Software Engineering*.

E. Yourdon and L. L. Constantine, *Structured Design*.

M. V. Zelkowitz, A. C. Shaw, and J. D. Gannon, *Principles of Software Engineering and Design*.

Resource Requirements (software):

- A program development environment will be essential.

Implementation Considerations and Concerns:

- It is imperative that students use the best available tools for version control, text editing, etc. Students will invariably draw on their experiences in actual system development rather than on what they have read or heard in lectures.
- Because the majority of learning in this course is by doing, a traditional course format may not be best. In addition to giving lectures, the instructor should spend time counseling teams, walking them through code-reading sessions, etc. A known problem with many software engineering courses taught in the past is that students become so involved with the project that they ignore the material in lecture. This has been partially addressed by including a large number of (hopefully) interesting topics not usually taught in software engineering courses.

8.2. Software Engineering Lab

Prerequisites: vary with the individual arrangement
SOFTWARE ENGINEERING [413]

Description: This course is intended to provide a vehicle for real-world software engineering experience. Students will work on existing software that is or will soon be in service. In a work environment, students will experience first-hand the pragmatic arguments for proper design, documentation, and other software practices that often seem to have hollow rationalizations when applied to code that a student writes for an assignment and then never uses again. Projects and supervision will be individually arranged.

Rationale: Software engineering issues arise in software that involves many months, many programmers, many versions, and many modules. These issues are extremely hard to raise in a one-semester course; they are easier to appreciate by working with real-world projects. This course is intended to provide an opportunity for training similar to a clinical practice course in a medical school. This will require closer cooperation between the industrial work site and the university than an ordinary work-study program would need. Evaluation of students will be shared between university faculty and the individual(s) managing them in the industrial organization.

Objectives: At the end of this course, a student will be able to:

- Apply software engineering principles to large, long-term projects
- Work effectively in a programming team

Ideas: This course will be the primary carrier of the following:

- Complexity of real-world systems
- Tools for dealing with that complexity

Implementation Considerations and Concerns: Getting good projects and good supervision.

- The people who serve as faculty for this course must be selected carefully.
- We must be careful not to have this become a "mindless programming for credit" course; the students must work on challenging projects that will force them to work with practicing software engineers.

9. Course Plan: Large Project Management Emphasis

During the last decade, one of the authors (Tomayko) has frequently taught the introductory software engineering course organized in a project-intensive manner. One such offering, in 1986, was captured in an SEI technical report [123]. The present section contains information from the fall 1990 version of the course, which is available from the SEI as an educational materials package [124]. It is interesting to compare the two offerings, which are four years apart, to note changes in topics and emphasis. For example, the concept of software process assessment is absent in 1986; and although risk reduction activity was practiced in the earlier course, it was not highlighted as it was later.

9.1. Project Background

The project used in this case study is an interactive exhibit that teaches the fundamentals of orbital rendezvous. The exhibit is destined for installation in the expansion of the Kansas Cosmosphere and Space Center in Hutchinson, Kansas, and is also usable for "Space Camp" classes held there. Max Ary, the director of the Cosmosphere, requested this software because he wanted something that would be "counter-intuitive" to capture the interest of patrons, and orbital mechanics certainly qualified. This project fulfills the criteria outlined earlier in this report: it has a real customer and target audience; it represents a challenge for the students (who are forced to gain domain knowledge in fields unfamiliar to most of them—in this case orbital mechanics and exhibit design); and it can be completed in one semester. One factor that "drove" the eventual project team organization is that the Cosmosphere had not yet decided between two platforms. They planned to use donated equipment to run the software, either an IBM PC/PC-clone or Apple Macintosh. This meant that two versions of the software needed to be built from one set of requirements and one design. From the standpoint of the customer, this was an advantage in that cosmetic factors, such as the quality of the graphics, could be used to influence the eventual hardware decision.

9.2. Syllabus

The syllabus for this course organization takes into consideration the following factors and topics:

Textbooks: Pressman's text contains almost enough information for a project-intensive course, with some deficiencies easily moderated by handouts. These deficiencies lie largely in the area of formal methods and newer topics such as process assessment. The book is not read in order from front to back because the topics are not in the precise order necessary to support the project (or any project, for that matter): the discussion of configuration management is in Chapter 15 of 15 chapters, somewhat late to be of help during the development process. Since the course project was related to space flight, the NASA standard for software development was used, with the students following the mandated outlines for the deliverable documentation.

This is the supplementary bibliography:

- [Brooks87] Frederick P. Brooks, Jr. "No Silver Bullet: Essence and Accidents of Software Engineering." *IEEE Computer*, 20, 4 (April 1987) 10-19.
- [Brooks75] Frederick P. Brooks, Jr. "The Surgical Team." Chapter 3 of *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975.
- [Coh86] B. Cohen, W. T. Harwood, M. I. Jackson. *The Specification of Complex Systems*, Chapter 7. Addison-Wesley, Wokingham, England, 1986.
- [Fagan76] M. E. Fagan. "Design and Code Inspections to Reduce Errors in Program Development," *IBM System Journal*, 3 (1976), 182-211.
- [Han76] S. L. Hantler, J. C. King. "An Introduction to Proving the Correctness of Programs." *Computer Surveys*, 8, 3 (September 1976) 333-353.
- [Har86] K. E. Harvey. "Summary of the SEI Workshop on Software Configuration Management." Software Engineering Institute Technical Report, CMU/SEI-86-TR-5, ADA 178770 (December 1986) Pittsburgh, PA, Carnegie Mellon University.
- [Humphrey88] W. S. Humphrey. "Characterizing the Software Process: A Maturity Framework." *IEEE Software*, 5, 2 (March 1988) 73-79.
- [JPL75] Jet Propulsion Laboratory. *Functional Requirement Viking Orbiter 1975 Flight Equipment Computer Command Subsystem Software,* (Insert in Orbital Design Book 612-2) California Institute of Technology, Pasadena, California, No. VO75-4-2005-2A, (17 January 1975).
- [JPL74] ---. *Viking 75 Orbiter Computer Command Subsystem Flight Software Design* Description, 612-28 Viking Orbiter Control Document, California Institute of Technology, Pasadena, California, Vol. I, Revision B (20 November 1974) 3-228–234, A-134–137.
- [Mills83] Harlan D. Mills. "Chief Programmer Teams: Techniques and Procedures". Article 8 in *Software Productivity*, Little, Brown and Company, 1983.
- [NASA86] National Aeronautics and Space Administration, Office of Safety, Reliability, Maintainability and Quality Assurance. *Software Development Standards, Data Item Descriptions,* Version 3.0 (15 October 1986).

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. Our method is to make assignments sufficiently well delineated that individuals are responsible for one thing or another. The team organization, with its multiple job categories, also helps define evaluation criteria. The remaining 50% is divided into 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 on grades.

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 time allotted for the final exam, usually three hours, is set aside for a final presentation of the substance of the project. This presentation 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. Preliminary and detailed design is scheduled for about five weeks. Coding is given two weeks, followed by a compressed testing and validation period, the compression made possible by division of labor and early quality assurance activity such as code inspections. Milestones are marked by in-class walkthroughs and reviews of all or part of the documents. These present an opportunity to demonstrate walkthrough, review, and inspection techniques.

3. Class presentations: Lectures and discussions are scheduled last. It is barely possible to stay half a life-cycle stage ahead of the project. The list of topics on this syllabus is ambitious but quite attainable.

Here is a brief description of the topics embodied in the individual class meetings:

1. **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. The software process maturity model and its implications for assessing the effectiveness of the total software development effort within a particular organization. [Brooks88; Pressman, Ch. 1]

2. **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. [Brooks75; Humphrey88; Mills83; NASA86.] (Note that the appropriate Data Item Descriptions from the NASA Standard are discussed when their topics are presented in class.)

3. **Requirements Engineering**

   - How requirements are determined. Interactions with customers, marketing, and development organizations. Stating requirements and developing the requirements document. [Pressman, Ch. 2, Ch. 4.1-4.4]

4. **Risk Reduction**

   - The concept of risk reduction as applied to software development. The use of prototypes. (Note that this particular offering of the course required the development of a functional prototype to reduce risk in two areas: the user interface and the calculation of orbital mechanics. The customer gave feedback on the interface to influence the final design.) [Pressman, Ch. 4.5-4.8]

5. **Controlling Disciplines: Quality Assurance 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. [Harvey86; Pressman Ch. 12, Ch. 15.7-15.8]

6. **Cost, Size, and Manpower Planning**

   - These topics relate to project management. 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. [Pressman, Ch. 3]

7. **Specification Techniques**

   - Formal specification tools and techniques such as on-line tools and data flow diagrams. Functional specification development. [Cohen86; Pressman, Ch. 5]

8. **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. [Pressman, Ch. 7-10]

9. **Design Representation**

   - Using structure charts, HIPO charts, data flow diagrams, pseudocode, and other tools in implementing designs. [Pressman, Ch. 6]

10. **Structured Programming and Implementation Considerations**

    - Review of the concepts of structured programming. Discussion of Bohm and Jacopini paper [20]. Applying structure to non-structured tools such as assembly languages. Coding considerations in FORTRAN and COBOL versus Pascal and Ada. [Pressman, Ch. 11]

11. **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. Code inspections. [Fagan76; Pressman, Ch. 13-14]

12. **Verification and Validation**

    - Formal verification, concept of validation and acceptance testing. Development of validation suites. Automated testing. [Pressman, Ch. 14; Han76]

13. **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. System description languages and their use. [Pressman, Ch. 15.1-15.6]

14. **User Documentation**

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

9.3. Flow of the Course

The students are introduced to the project on the first day. Within a week, the project team is organized by the instructor and includes roles in all phases of development: designers, implementors, quality assurance persons, configuration managers, and the important role of principal architect, who is responsible for overseeing the key development phases. The students actually apply for these "jobs" and are interviewed for them by the instructor.[4]

The analysis and design team begins the task of establishing the requirements for the user interface and for the physics. The technical support teams (configuration management, quality assurance, independent verification and validation) organize themselves and write their respective operation plans. The prototype is built and demonstrated to the customer. Final specification documentation is completed, allowing the independent verification and validation team to begin constructing the acceptance test suite and the documentation team to write the user manual. The architectural-level design with all interfaces is written, so the test and evaluation teams for both the PC and Mac versions can begin specifying the integration tests. The detailed design is completed, so the implementors can code. As these milestones are reached, the configuration control board meets regularly to decide if documentation is ready to be placed under configuration management. These meetings also include analysis of change requests and discrepancy reports generated against controlled items. As the coding progresses, the quality assurance group conducts code inspections and audits the development and testing process. When the implementation is complete and integrated, the acceptance tests are executed, the user manual is evaluated against the software, and the two versions are prepared for shipment. The final review is held, followed by a celebration at the nearest pub.

[4]We are grateful to Prof. Linda Northrop of SUNY-Brockport for the idea of interviewing the students. She used the 1987 technical report as the basis for a course, innovated this idea, and fed it back to us [94]. It serves as an excellent method of quickly learning something about the students' backgrounds, as they are required to show up with a resume. Previously, the roles were assigned by the instructor based on pseudo-random choice and instinct.

10. Course Plan: Even Balance of Management, Engineering, and Tools

This version of the introductory software engineering course was taught in the fall of 1989 by one of the authors (Shaw) using a project with a real deliverable for a real client. This case study describes the background and organization of the course and sketches the project. The description is drawn from a complete report on the course, including the supplemental materials and course handouts, prepared for the SEI Software Engineering Curriculum Project [25].

We decided that we could make the characteristics of software systems more vivid by choosing a project whose result could benefit some group on campus, preferably the campus computing community at large. We polled the local community for suggestions and chose a proposal from the Information Technology Center (the group that developed Andrew, the campus-wide computing system). They proposed combining existing software facilities to create a bridge between electronic mail and facsimile transmission provided by a special fax board running in a personal computer. The students succeeded in developing a running prototype, which they demonstrated in a formal presentation and acceptance test at the client's site.

In planning the lecture component of the course, we examined most of the popular textbooks. None of them suited our conviction that the course should emphasize the technical substance that should lie behind design decisions. We decided that our students should be able to read material at the level of *IEEE Software* and *IEEE Computer* and, further, that the material would be more vivid in the words of the original authors. We therefore decided to teach the course from a collection of readings instead of a textbook. We had to allow lecture time for enough material on project planning, organization, and management to run the course project, as well as for introducing tools and technical information in support of the project. After making these allowances, we wound up with a roughly even balance of technical, management, and tool topics, with a few lectures left to address the character of the profession. The reading list appears below.

To discourage the students from devoting all their attention to the project (which often leads to ignoring material from lectures that would help with the project), we allocated 60% of the course grade to the group grade for the project and 40% to individual effort on the readings. We encouraged students to keep up with the readings by giving short quizzes and homework assignments on a daily basis instead of giving one or two exams.

10.1. Syllabus

Objectives: Upon completion of this course, a student should:

- Understand the difference between a program and a software product.
- Be able to design and implement a module that will be integrated in a larger system.

Each student will have demonstrated the ability to:

- Work as a member of a project team, assuming various roles as necessary.
- Create and follow project plans and test plans.
- Create the full range of documents associated with software products.
- Read and understand papers from the software literature.

Computing: The project will be implemented as a service in Andrew (the campus computing facility). If you don't have an Andrew account, we'll help you get one. The course bulletin board is academic.cs.15-413 and various sub-bboards. Subscribe to them.

Grading:

- *Project* (60%): 8% for each phase: requirements, design, project plan, functional specification, implementation, unit testing, integration, and client acceptance.

  Special incentive: if a complete product (specifications, project plan, internal and user documentation, and working code) with core functionality is delivered to the client as a joint effort of the course, all students will receive at least 55 points of 60 for the project.

- *Lectures* (40%): 2% for each of 22 lectures: 1 point for short quiz on main point of the reading, 1 point for 1-2 page homework on main points of class discussion.
- *Instructors' evaluation:* adjustment of up to 5%.
- *Standards:* These were set at the beginning of the course to reduce competition for curved grades.

  A: 90+
  B: 80-89
  C: 70-79 (including at least 25 points from lectures and 40 points from project)
  D: 65-69 (or 70-79 unbalanced)
  R: less than 65

Lecture Topics [28 lectures, 22 with readings]: Because the project required certain information by certain developmental stages, the order of presentation was slightly different from the topical organization. In addition, the topics on the nature of the profession did not come simultaneously.

1. **Introduction**

   - *Course organization* (8/31). The nature of software engineering. A brief sketch of its history. Products versus systems. Introduction to project. Reading after class: Brooks 75 Ch 1.

2. **Life Cycle, Requirements, Specifications, Documentation**

   - *Requirements* (9/7). Determining what the client actually wants. Expressing it precisely. Notations for requirements. Reading: Brooks75 Ch 6; Davis82.

- *Life cycles* (9/12). The stages a software project goes through, from conception and development to maintenance and retirement. Models for this life cycle. How well the models match reality. Reading: Brooks75 Ch 13; Davis88.

- *Documentation* (9/14). Retention and presentation of the information that is part of a software product but not explicit in the code. Reading: Brooks75 Ch 10, 15.

3. **Tools and Standards**

- *Fax formats and protocols* (9/19). Information about fax formats and communication protocols that will be needed for the project. Reading: McComb89; CCITT Group 3 and Group 4.

- *Standards; Andrew* (9/21). Role of standards in software. Information about the Andrew editor and libraries that will be needed for the project. Reading: Poston84-85.

- *Configuration management and version control* (10/12). Consistency among versions of subcomponents. Automation of system construction. Baselining and version control. Reading: Feldman79; Tichy82.

4. **Management**

- *Project Planning* (9/26 and 9/28). Justifying projects. Making them fit within existing systems. Project organization and milestones. Reading: Brooks75 Ch 2, 3, 14; Davenport89; Fairley86.

- *Estimation and Tracking* (10/5 and 10/10). Predicting size of product. Estimating time required to create it. Models and statistics for these predictions. How well the models work. Reading: Brooks75 Ch 7, 8, 9; Myers78; Myers89; AdaIC89.

- *Verification and validation* (10/17). Techniques for gaining confidence that software works. Reading: Wallace89.

5. **Software Design**

- *Abstraction* (10/26). Role of abstraction in software engineering. Increasing abstraction size as index of growth. Reading: Shaw84.

- *System Design* (10/31 and 11/7). Conceptual integrity. System-level design techniques. Survey of design methodologies. Reading: Brooks75 Ch 4, 5; Lampson84; Bergland81.

- *Software Structures* (11/2). System-level abstractions for software. Reading: Shaw89.

- *Software Reuse* (11/9). Not reinventing the wheel. Reading: Prieto-Diaz87.

6. **Back End**

- *Programming Environments* (11/16). Tools and environments to support software development. Reading: Brooks75 Ch 12; Kernighan81; Dart87.

- *Testing* (11/21). Planning and executing a testing strategy. Reading: Howden85.

- *Maintenance* (11/30). Life after initial release. Fixing design errors, adding new features. Reading: Brooks75 Ch 11; Schneidewind87.

7. **The Software Engineering Profession**

- *The engineering component of software engineering* (10/3). Comparison of software engineering to older engineering disciplines. Lessons software engineering should draw from this comparison.

- *Status of the profession* (10/24). Concerns and prospects of the software engineering profession. Reading: Musa85.

- *The work force and the job market* (11/14). What it's like to be a practitioner in software. Panel discussion with representatives from big software development, startup software, and application software companies, and a professional recruiter.

- *Intellectual property issues* (12/5). Kinds of intellectual property protection. Ownership of results produced by programs. Reading: Legal Task Force 84, Gemignani85.

8. **Project Discussions**

- *Requirements for project* (9/5). Client presents needs and answers questions.

- *Client's design review* (10/19). Presentations of design. Opportunity for mid-course correction.

- *Internal review of project* (11/28). Class discussion of project: progress, lessons learned.

- *Final presentations to client* (12/7). Demonstration, acceptance test.

Textbooks and Readings:

- Frederick P. Brooks. *The Mythical Man-Month*. Addison Wesley, 1975, reprinted 1982.

- Marvin V. Zelkowitz. *Selected Reprints in Software, Third Edition*. Computer Society Press, 1987.

References to *Selected Reprints* are papers reprinted in Zelkowitz.

- [Ada IC89] Ada IC staff. "Test case study: estimating the cost of Ada software development." *Ada Information Clearinghouse Newsletter*, March 1989, pp. 4-6.

- [Bergland81] G. D. Bergland. "A guided tour of program design methodologies." *Selected Reprints*, p. 28.

- [Brooks75] Frederick P. Brooks Jr. *The Mythical Man-Month*. Addison-Wesley, 1975, reprinted 1982.

- [CCITT Group 3] CCITT. "Standardization of group 3 facsimile apparatus for document transmission."

- [CCITT Group 4] CCITT. "Facsimile coding schemes and coding control functions for group 4 facsimile apparatus."

- [Dart87] Susan A. Dart et al, "Software development environments," *IEEE Computer*, November 1987, pp. 18-28.

- [Davenport89] Thomas H. Davenport. "The case of the soft software proposal." *Harvard Business Review*, May-June 1989, pp. 12-24.

- [Davis82] A. M. Davis, "The design of a family of application-oriented requirements languages." *Selected Reprints*, p. 20

- [Davis88] A. M. Davis et al. "A strategy for comparing alternative software development life cycle models." *IEEE Transactions on Software Engineering*, October 1988, pp. 1453-1461.

- [Fairley86] Richard E. Fairley. "A guide for preparing software project management plans." Wang Institute Tech Report TR-86-14, November 1986.
- [Feldman79] S. I. Feldman, "Make: a program for maintaining computer programs" *Software Practice and Experience*, April 1979, pp. 255-265.
- [Gemignani85] M. C. Gemignani. "Who owns what software produces?" *Selected Reprints*, p. 121.
- [Howden85] W. E. Howden. "The theory and practice of functional testing." *Selected Reprints*, p. 258.
- [Kernighan81] B. W. Kernighan et al. "The Unix programming environment." *Selected Reprints*, p. 287.
- [Lampson84] B. W. Lampson. "Hints for computer system design." *IEEE Software*, January 1984, pp. 11-28.
- [Legal Task Force84] Task Force on Legal Aspects of Computer-Based Technology. "Protection of computer ideawork—today and tomorrow." *Selected Reprints*, p. 126.
- [McComb89] Gordon McComb et al. "The fax factor." *MacUser*, August 1989, pp. 149-160.
- [Musa85] J. D. Musa. "Software engineering: the future of a profession." *Selected Reprints*, p. 2.
- [Myers78] W. Myers. "A statistical approach to scheduling software development." *Selected Reprints*, p. 53.
- [Myers89] W. Myers. "Allow plenty of time for large-scale software." *IEEE Software*, July 1989, pp. 92-99.
- [Poston84-85] Robert M. Poston. "Software standards." Three columns on software standards from *IEEE Software*, January 1984, May 1985, September 1985.
- [Prieto-Diaz87] R. Prieto-Diaz et al. "Classifying software for reusability." *Selected Reprints*, p. 94.
- [Schneidewind87] N. F. Schneidewind. "The state of software maintenance." *IEEE Transactions on Software Engineering*, March 1987, pp. 303-310.
- [Shaw84] M. Shaw. "Abstraction techniques in modern programming languages." *Selected Reprints*, p. 232
- [Shaw89] M. Shaw. "Larger-scale systems require higher-level abstractions," *Proc. Fifth Int'l Workshop on Software Specification and Design*, May 89, pp. 143-146.
- [Tichy82] W. F. Tichy, "Design, implementation, and evaluation of a revision control system." *Proc. 6th Int'l Conference on Software Engineering*, 1982, pp. 58-67.
- [Wallace89] D. R. Wallace et al. "Software verification and validation: an overview." *IEEE Software*, May 1989, pp. 10-17.

10.2. Project

The main goal of the project was to give students a realistic view of the problems involved in the manufacture of a complex software system. Our intention was to avoid the "toy program" approach and make the project as realistic as possible. The project was supposed to provide a vehicle for the students to get hands-on experience with both technical and managerial aspects of building a large software system in cooperation with others.

The project had to be of realistic size, something that could be done by about 20 students within one semester. This was a primary consideration in our selection of the proposal to connect electronic mail to fax transmissions, a project we called Workstation Fax. A task of this magnitude could not be done if the system had to be designed and implemented from scratch. Identification of existing software and hardware for reuse was therefore a major part of our project preparation. One of the attractive properties of the Workstation Fax proposal was that it identified several existing components and proposed a project that combined these elements to obtain the final product. That is, the students produced the "glue" to connect existing components, not stand-alone software. The main component to be reused was the Andrew mail message system.

In Workstation Fax the user sends and receives faxes via the Andrew mail facility without the need for producing hard copy. Sending a fax image from an Andrew workstation involves conversion of a text or Andrew raster image into Group 3 fax format. As part of the Andrew project, the ITC has built tool kits for dealing with a variety of raster image conversions, including Group 3 fax. This tool kit provided the underlying routines for manipulating fax images. Receiving a fax image via e-mail is more difficult because the delivery information on the cover sheet is not in digital form. It would be unrealistic to expect this project to include software to interpret the wide variety of cover page formats, including the handwriting often used to provide routing. To deal with this, we decided to route incoming faxes manually by reusing the Andrew bulletin board facility: the incoming fax is posted on a special Andrew bulletin board and routed from there by a human to the final destination. To avoid requiring a full implementation of the Group 3 fax protocol (which would have been unreasonably difficult), we purchased a commercially available fax board that runs on an IBM PC. The Andrew file system provides mechanisms for file transfer to and from this PC.

The emphasis in Workstation Fax was on functionality and usability; performance was secondary. Receiving or sending fax images takes a matter of minutes; we therefore assumed that a system latency (the time between sending a fax and receiving it) on the order of 15-30 minutes would be acceptable, assuming no additional delays in the mail system.

11. Administering a Project Course

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. Giving advice and guidance, acquiring resources, and sitting in on reviews, walkthroughs, and other meetings demand a considerable time investment. However, for both student and teacher, the rewards of this course are great as well. The feeling of accomplishment, the experience of learning to work together as a team, the use of skills previously exercised only in limited situations, all contribute to an exceptionally positive learning experience. Our former students often tell us that the only college friends they keep in touch with are their teammates from the software engineering course. Their positive feelings are 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 I see my spouse and children again before Christmas?

11.1. Preliminary Preparation

Getting ready to teach this course begins several months in advance. The instructor needs 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.

11.1.1. Class Size

A class of 15 to 20 students is enough critical mass to cause the communications and configuration control explosion that the instructor is trying to engineer. Actually, any larger number 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 compromises the instructor’s ability to do the job. Using assistants often backfires since they also require management. Frankly, the main reason for keeping the class to 20 or so is that no more people are needed to successfully do a project that has a four-month development period. In one case, the actual number of students who

showed up the first day was 38. By the next class, eight had disappeared, presumably because they had read the syllabus (a good reason for handing one out right away). The remaining 30 were too many for the actual project, so the instructor 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 (coders are needed for both languages, for example), thus creating work for more students.

11.1.2. Choosing a Project

In a project-intensive course, it is very important to come up with an interesting project. The requirement for "interesting" 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 tedium. For example, some of our 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), a mission planning simulator for a manned research station on Mars, and an orbital rendezvous simulator as a teaching exhibit for a science museum. 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 for technical expertise balanced things out. Also note that each of these projects was unlikely to have "experts" in the class that outshine the other students in requirements analysis. Choosing a project area in which students do not have much experience creates a situation in which the students have to interact with the user extensively during the requirements definition stage, thus developing a bond that can provide motivation throughout the project.

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 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 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 the students. Some instructors have created continuing projects based on this principle. Walt Scacchi of the University of Southern California had many undergraduate and graduate students in software engineering courses participate in creating programming tools. This class of projects survived for many years.

11.1.3. Choosing the Development Environment

After the project and class size are determined, the next step is choosing the development environment. Sometimes this is a trivial matter, when only one environment exists. If PCs or mainframe systems are all that are available, the project may have to be adapted. If PCs are available but the class size dictates that they will be overused, the class might have to stay

on the mainframes. Sometimes the customer will have a specific target machine. For instance, the football team whose scouting system we developed had a donation from a booster—an IBM System 23 with only BASIC. Our classes have used PCs, mainframes, MicroVax workstations, Macintosh IIs, and superminis, all without having to abandon good software engineering practice. Furthermore, remember that only a small percentage of the class will be actually coding. During the choreographer's assistant project, we were limited to developing code on two graphics-equipped PCs, and this was no problem. Project documentation, however, had to be maintained on a central mainframe system.

A more important consideration is the availability of tools. Needed, at least, are 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 or similar tools, configuration management must be done manually.

The e-mail or electronic bulletin board facility 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 as most of the development information and out-of-class instruction can be transmitted by electronic mail. One of the authors handled about 1,400 messages relating to the course during a recent semester. Fortunately, most mainframe and minicomputer systems now come with some sort of communications. Obviously, however, non-networked PCs cannot be used for mail. That is another reason to keep documentation and mail on some central system even if the target is a PC.

11.2. Staffing

A project-intensive course in software engineering requires considerable time and attention to detail on the part of the course staff. A list of things to be done by the instructor includes:

- Preparing and presenting lectures.
- Preparing and grading quizzes.
- Preparing and grading homework.
- Designing the project and anticipating where it can go astray.
- Writing and revising project documents.
- Setting up common procedures (version control, document templates).

- Acquiring tools, components, and associated documentation.
- Coordinating with the client.
- Troubleshooting the lab.

If a department provides teaching assistants, they can handle many of these tasks. If not, the instructor may be able to get graduate students to volunteer as group leaders and mentors. If the course enrollment includes graduate students with more experience than the undergraduates, they can take on these roles. Another potential source of assistance is university staff, particularly if the project will yield a system of general utility. Programming staff are sometimes delighted to have the opportunity to get some teaching experience.

11.3. Coordination Between Lectures and Project

In a course organized around a project, synchronization between the class lectures and the project phases is very important. If synchronization is done well, students can instantiate class concepts almost immediately in the project and can apply the project experience in class.

Most of the coordination takes place in pre-course planning, when lecture topics are scheduled. Lectures on activities most directly related to the project need to be scheduled at the times they will be most helpful, and any remaining lectures can then be arranged in the best available approximation to a coherent thread. Unfortunately, this plan requires about half of the lectures to be given in the first three weeks of the course.

Synchronization is especially hard to achieve in the early phases of the project. The students are not yet familiar with certain concepts which are already needed for the project. Furthermore, it is not possible to have the students apply all the concepts taught in class. One solution to this problem would be to teach the course in two semesters. All the software engineering concepts would be taught in the first semester and then applied to the project in the second. Despite synchronization problems, however, we believe it is better to teach the course in one semester. We found that whenever the project demanded some knowledge before it was taught, the students were much more motivated when we covered the material in the lecture!

Another way to keep the lectures coordinated with the project is to assign homework questions in which students must apply lecture material to the project.

11.4. Credit and Grading Policy

A course of this kind presents several special problems, including:

- Grading team efforts
- Fostering cooperation rather than competition
- Keeping lectures relevant
- Getting the readings read

We will discuss each problem in turn, describing how we addressed it. The common thread of our solutions is to be explicit about our objectives and align the incentives (primarily grades) with the behaviors we wish to encourage. This does, of course, force us to grade what's important rather than what's easy to grade.

11.4.1. Grading Team Efforts

Grading a team of students is different from grading students individually. With group grades there is the danger that very active students might carry the load while others get a "free ride." Careful project management is the best way to avert this problem. Part of learning to work in a team is learning how to share the load. It's always possible that a problem student will require special handling, but a good policy is to begin by assuming that groups can work together.

An alternative is to assign individuals tasks that are specific enough for assigning individual, rather than group, grades.

11.4.2. Fostering Cooperation Rather Than Competition

Like many students, ours are competitive. This competition is often grade-directed, and uncertainties about class standings can distract students from learning. Even worse, students are (or claim to be) accustomed to courses graded "on a curve"—with a limit on the number of "A" and "B" grades awarded. This approach inhibits cooperation and even leads to counterproductive behavior that would lower some other student's grade.

Because a project course depends critically on cooperation among students, we addressed this problem directly: we defused the uncertainty of the grading curve by publishing the grading scale at the beginning of the semester and declaring that there would not be a curve. In addition to assigning group grades (which promotes cooperation within groups), we provided a completion incentive: if the project passed the acceptance test through the efforts of the class as a whole, every student would receive at least 55 of the 60 project points.

11.4.3. Keeping Lectures Relevant

When the grade in a course depends primarily on project work, students tend to spend their time on the project instead of the lecture and associated readings. (This is true in programming courses in general; in extreme cases we've seen students so focused on making progress on a project with brute force that they wouldn't pay attention to the lectures telling them how to solve the problems easily.)

We used several ways to convince students that the lectures were relevant. First, we committed 40% of the course grade to individual performance in the lecture portion of the course. This is commensurate with our assessment of the appropriate balance of time and content; happily it also helps reduce apprehension about the vulnerability of one's own grade to the vagaries of other students. Second, we scheduled the material for presentation as nearly as possible at the time students would need it for the project. Finally, we usually wrote a homework assignment that required students to explain a connection between the lecture and the project. In more than one case, we then incorporated their answers into our next lecture.

11.4.4. Getting the Readings Read

In any course, assigned readings are often postponed—usually until the night before the test. We have been daunted by the prospect of students trying to do the reading in this way.

Thus, we gave a 5-minute quiz at the beginning of every class with an assigned reading (about 22 of the 28 classes). The quiz was easy and intended to determine whether the students had captured the main point of the reading. For the most part, the quizzes showed the students to be doing the reading. An added benefit was that the lectures could then assume the reading as shared context between the instructor and the students—so the lectures could provide motivation, context, and evaluation rather than just repeating the substance of the reading.

Related SEI Publications

A Software Engineering Project Course with a Real Client, Bernd Bruegge, John Cheng, Mary Shaw, CMU/SEI-91-EM-4

> This SEI educational materials package fully documents the project course styles described in Section 9 of this report.

Materials to Support a Project-Intensive Introduction to Software Engineering, James A. Tomayko, CMU/SEI-91-EM-6.

> This SEI educational materials package fully documents the project course styles described in Chapter 10 of this report.

Teaching a Project-Intensive Introduction to Software Engineering, James A. Tomayko, CMU/SEI-87-TR-20, DTIC: ADA200603.

> 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. Other models of course organization are also discussed. Additional materials used in teaching the course and samples of student-produced documentation are also available.

Software Engineering Education Directory, CMU/SEI-91-TR-4, DTIC: ADA223704.

> Data for the current report was drawn from this directory. The directory provides information about software engineering courses and software engineering degree programs offered by colleges and universities, primarily in the United States. The most recent version of the directory is CMU/SEI-91-TR-9.

Software Maintenance Exercises for a Software Engineering Project Course, C. Engle, G. Ford, and T. Korson, CMU/SEI-89-EM-1, DTIC: ADA235779

> This report provides an operational software system of 10,000 lines of Ada and several exercises on that system. Concepts such as configuration management, regression testing, code reviews, and stepwise abstraction can be taught with these exercises. Diskettes containing code and documentation may be ordered for $10.00. Please request either IBM PC or Macintosh disk format.

Software Project Management, James E. Tomayko, The Wichita State University, Harvey K. Hallman, Software Engineering Institute, SEI-CM-21, DTIC: ADA237048, July 1989.

> Software project management encompasses the knowledge, techniques, and tools necessary to manage the development of software products. This curriculum module discusses material that managers need to create a plan for software development, using effective estimation of size and effort, and to execute that plan with attention to productivity and quality. Within this context, topics such as risk management, alternative life-cycle models, development team organization, and management of technical people are also discussed.

How to Order SEI Reports

If you are an SEI affiliate, you may order all reports directly from the SEI. To request your copy, please submit your written request, accompanied by a mailing label with your return address, to:

Software Engineering Institute
ATTN: Publications Requests
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Reports that have DTIC numbers (with the prefix ADA) are available from two national distribution centers: the Defense Technical Information Center (DTIC) or the National Technical Information Service (NTIS). If you are not an SEI affiliate, please request copies of these reports by contacting either organization directly.

DTIC Defense Technical Information Center
ATTN: FDRA
Cameron Station
Alexandria, VA 22304-6145

NTIS National Technical Information Service
U.S. Department of Commerce
Springfield, VA 22161-2103

Acknowledgements

This report relies heavily on our prior work. Tomayko's comparison of formats [123] supplied the bulk of Section 3, much of Section 11, and the case study in Section 9. Bruegge, Cheng, and Shaw's description of another offering [25]. provided much of Chapter 11 and the case study of Section 10. Section 8 was taken from Shaw's earlier curriculum design effort [113]. Objective survey data was taken from the Software Engineering Institute's survey of software engineering programs [84]; the more informal survey of organizations has been done over a period of seven years by Tomayko, during visits to computer science departments as an ACM Distinguished National Lecturer and as part of the SEI Software Engineering Curriculum Project.

References

1. Abbott, Russell J. *An Integrated Approach to Software Development.* John Wiley & Sons, Inc., New York, New York, 1986.

2. ACM Curriculum Committee on Computer Science. "Curriculum 78: Recommendations for the Undergraduate Program in Computer Science". *Communications of the ACM 22*, 3 (March 1979), 147-166.

3. Aron, Joel D. *The Program Development Process.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1974.

4. Arthur, Lowell Jay. *Software Evolution: The Software Maintenance Challenge.* John Wiley & Sons, Inc., New York, New York, 1988.

5. Babich, Wayne A. *Software Configuration Management: Coordination for Team* Productivity. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

6. Backhouse, Roland C. *Program Construction and Verification.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1986.

7. Barstow, David R., Shrobe, Howard E., and Sandewell, Erik, editors. *Interactive Programming Environments.* McGraw-Hill, New York, New York, 1984.

8. Basili, Victor R., editor. *Tutorial on Models and Metrics for Software Management and Engineering.* IEEE Computer Society, New York, New York, 1980.

9. Bauer, Friedrich Ludwig, editor. *Advanced Course on Software Engineering.* Springer-Verlag, Berlin, Germany; New York, New York, 1973.

10. Beck, Leland L. *System Software: An Introduction to Systems Programming, 2nd* edition. Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

11. Beizer, Boris. *Software System Testing and Quality Assurance.* Van Nostrand Reinhold, New York, New York, 1984.

12. Beizer, Boris. *Software Testing Techniques, 2nd edition.* Van Nostrand Reinhold, New York, New York, 1990.

13. Bell, Doug, Morrey, Ian, and Pugh, John. *Software Engineering: A Programming* Approach. Prentice-Hall International, Englewood Cliffs, New Jersey, 1987.

14. Bentley, Jon Louis. *Writing Efficient Programs.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1982.

15. Bentley, Jon Louis. *Programming Pearls.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

16. Bergland, Glenn D. and Gordon, Ronald D., editors. *Tutorial: Software Design Strategies, 2nd edition.* IEEE Computer Society, Institute of Electrical and Electronics Engineers, Los Angeles, California; New York, New York, 1981.

17. Bersoff, Edward H., Henderson, Vilas D., and Siegel, Stanley G. *Software Configuration Management: An Investment in Product Integrity.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1980.

18. Birrell, N. D. and Ould, Martyn A. *A Practical Handbook for Software Development.* Cambridge University Press, Cambridge [Cambridgeshire], England; New York, New York, 1985.

19. Boehm, Barry W. *Software Engineering Economics.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1981.

20. Bohm, C. and Jacopini, G. "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules". *Communications of the ACM 8*, 5 (May 1966), 366-371.

21. Booch, Grady. *Software Engineering with Ada, 2nd edition.* Benjamin/Cummings Publishing Company, Menlo Park, California, 1987.

22. Booch, Grady. *Software Components with Ada: Structures, Tools, and Subsystems.* Benjamin/Cummings Publishing Company, Menlo Park, California, 1987.

23. Brockmann, R. John. *Writing Better Computer User Documentation: From Paper to Online.* John Wiley & Sons, Inc., New York, New York, 1986.

24. Brooks, Frederick P., Jr. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1982.

25. Bruegge, Bernd, Cheng, John, and Shaw, Mary. *A Software Engineering Project Course with a Real Client.* Software Engineering Institute, Carnegie Mellon University, 1991. CMU/SEI-91-EM-4.

26. Bryan, William, Chadbourne, Christopher, and Siegel, Stanley G. *Tutorial: Software Configuration Management.* IEEE Computer Society, Institute of Electrical and Electronics Engineers, Los Alamitos, California, 1980.

27. Bryan, William L. and Siegel, Stanley G. *Software Product Assurance: Techniques for Reducing Software Risk.* Elsevier Science Publishers, New York, New York, 1988.

28. Buckley, Fletcher J. *Implementing Software Engineering Practices.* John Wiley & Sons, Inc., New York, New York, 1989.

29. Budde, Reinhard, et al., editors. *Approaches to Prototyping.* Springer-Verlag, Berlin, Germany; New York, New York, 1984.

30. Chandrasekaran, B. and Radicchi, Sergio. *Computer Program Testing.* North-Holland Publishing Company, Amsterdam, Netherlands; New York, New York, 1981.

31. Charette, Robert N. *Software Engineering Environments: Concepts and Technology.* Intertext Publications, New York, New York, 1986.

32. Chow, Tsun S., compiler. *Tutorial on Software Quality Assurance: A Practical* Approach. IEEE Computer Society, Silver Spring, Maryland, 1985.

33. Cohen, Bernard, Harwood, William T., and Jackson, Melvyn I. *The Specification of Complex Systems.* Addison-Wesley Publishing Company, Wokingham, England; Reading, Massachusetts, 1986.

34. Connor, Denis. *Information System Specification and Design Road Map.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

35. Conte, Samuel Daniel, Dunsmore, H. E., and Shen, V. Y. *Software Engineering Metrics and Models.* Benjamin/Cummings Publishing Company, Menlo Park, California, 1986.

36. Darnell, Peter A. and Margolis, Philip E. *Software Engineering in C.* Springer-Verlag, New York, New York, 1988.

37. Davis, William S. *Tools and Techniques for Structured Systems Analysis and Design.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

38. Davis, Alan Michael. *Software Requirements: Analysis and Specification.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1990.

39. Deimel, Lionel E., editor. *Software Engineering Education.* Springer-Verlag, Berlin, Germany, 1990.

40. DeMarco, Tom. *Structured Analysis and System Specification.* Yourdon Press, Englewood Cliffs, New Jersey, 1979.

41. DeMarco, Tom. *Concise Notes on Software Engineering.* Yourdon Press, Englewood Cliffs, New Jersey, 1979.

42. DeMarco, Tom. *Controlling Software Projects: Management, Measurement and* Estimation. Yourdon Press, New York, New York, 1982.

43. DeMillo, Richard A., et al. *Software Testing and Evaluation.* Benjamin/Cummings Publishing Company, Menlo Park, California, 1987.

44. Deutsch, Michael S. *Software Verification and Validation: Realistic Project* Approaches. Prentice-Hall International, Englewood Cliffs, New Jersey, 1982.

45. Deutsch, Michael S. and Willis, Ronald R. *Software Quality Engineering: A Total Technical and Management Approach.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1988.

46. Eliason, Alan L. *Systems Development: Analysis, Design, and Implementation, 2nd edition.* Scott, Foresman/Little, Brown Higher Education, Glenview, Illinois, 1990.

47. Fairley, Richard E. *Software Engineering Concepts.* McGraw-Hill, New York, New York, 1985.

48. Fairley, Richard E. and Freeman, Peter, editors. *Issues in Software Engineering Education.* Springer-Verlag, New York, New York, 1989.

49. Fisher, Allen. *CASE: Computer-Aided Software Engineering: Using the Newest Tools in Software Development.* John Wiley & Sons, Inc., New York, New York, 1988.

50. Ford, Gary A., editor. *Software Engineering Education.* Springer-Verlag, New York, New York, 1988.

51. Freedman, Daniel P. and Weinberg, Gerald M. *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products, 3rd edition.* Little, Brown, Boston, Massachusetts, 1982.

52. Freeman, Peter, editor. *Tutorial: Software Reusability.* IEEE Computer Society, Washington, DC, 1987.

53. Freeman, Peter and Wasserman, Anthony I. *Tutorial on Software Design Techniques, 4th edition.* IEEE Computer Society, Silver Spring, Maryland, 1983.

54. Gane, Chris and Sarson, Trish. *Structured Systems Analysis: Tools and Techniques.* McAuto, St. Louis, Missouri, 1982.

55. Gehani, Narain and McGettrick, Andrew D., editors. *Software Specification* Techniques. Addison-Wesley Publishing Company, Wokingham, England; Reading, Massachusetts, 1986.

56. Gibbs, Norman E., editor. *Software Engineering Education.* Springer-Verlag, Berlin and Heidelberg, Germany, 1989.

57. Gibbs, Norman E. and Fairley, Richard E., editors. *Software Engineering Education: The Educational Needs of the Software Community.* Springer-Verlag, New York, New York, 1987.

58. Gilb, Tom. *Software Metrics.* Winthrop Publishers, Cambridge, Massachusetts, 1977.

59. Gilb, Tom. *Principles of Software Engineering Management.* Addison-Wesley Publishing Company, Wokingham, England; Reading, Massachusetts, 1988.

60. Gilbert, Philip. *Software Design and Development.* Science Research Associates, Chicago, Ilinois, 1983.

61. Hansen, Hans-Ludwig. *Software Validation: Inspection - Testing - Verification* - Alternatives. Elsevier Science Publishers B.V., Amsterdam, Netherlands; New York, New York, 1984.

62. Hetzel, William C. *The Complete Guide to Software Testing, 2nd edition.* QED Information Sciences, Wellesley, Massachusetts, 1988.

63. Hoffman, Daniel. "An Undergraduate Course in Software Design". *Software Engineering Education*, Proceedings of the SEI Conference 1988, Fairfax, VA, Springer-Verlag, New York, New York, April 1988, pp. 154-168.

64. Horning, James J. "The Software Project as a Serious Game". *Software Engineering Education Needs and Objectives, Proceedings of an Interface Workshop*, New York, New York , 1976, pp. 71-77.

65. Humphrey, Watts S. *Managing the Software Process.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

66. Hunke, Horst, editor. *Software Engineering Environments.* North-Holland Publishing Company, Amsterdam, Netherlands; New York, New York, 1981.

67. Jones, Cliff B. *Software Development: A Rigorous Approach.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1980.

68. Jones, Capers. *Programming Productivity.* McGraw-Hill, New York, New York, 1986.

69. Kant, Elaine. "A Semester Course in Software Engineering". *ACM Software Engineering Notes 6*, 4 (August 1981).

70. Kendall, Penny A. *Introduction to Systems Analysis and Design: A Structured* Approach. Allyn and Bacon, Boston, Massachusetts, 1987.

71. Kendall, Kenneth E. and Kendall, Julie E. *Systems Analysis and Design.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1988.

72. Kernighan, Brian W. and Plauger, P. J. *The Elements of Programming Style, 2nd* edition. McGraw-Hill, New York, New York, 1978.

73. Kezsbom, Deborah S., Schilling, Donald L., and Edward, Katherine A. *Dynamic Project Management: A Practical Guide for Managers and Engineers.* John Wiley & Sons, Inc., New York, New York, 1989.

74. King, David. *Current Practices in Software Development: A Guide to Successful* Systems. Yourdon Press, Englewood Cliffs, New Jersey, 1984.

75. Kopetz, Hermann. *Software Reliability.* Springer-Verlag, New York, New York, 1979.

76. Kowal, James A. *Analyzing Systems.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1988.

77. Lamb, David Alex. *Software Engineering: Planning for Change.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1988.

78. Leventhal, Laura Marie and Mynatt, Barbee T. "Stalking the Typical Undergraduate Software Engineering Course: Results from a Survey". Issues in Software Engineering Education, NewYork, 1989, pp. 168-195.

79. Lewis, Theodore Gyle. *Software Engineering: Analysis and Verification.* Reston Publishing Company, Reston, Virginia, 1982.

80. Liskov, Barbara and Guttag, John. *Abstraction and Specification in Program* Development. MIT Press; McGraw-Hill, Cambridge, Massachusetts; New York, New York, 1986.

81. Londeix, Bernard. *Cost Estimation for Software Development.* Addison-Wesley Publishing Company, Wokingham, England; Reading, Massachusetts, 1987.

82. Marca, David. *Applying Software Engineering Principles.* Little, Brown, Boston, Massachusetts, 1984.

83. Martin, James and McClure, Carma L. *Structured Techniques: The Basis for CASE.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1988.

84. McSteen, Bill, Gottier, Brian, and Schmick, Mark, editors. *Software Engineering Education Directory*. Tech. Rept. CMU/SEI-90-TR-4, DTIC: ADA 223740, Software Engineering Institute, Carnegie Mellon University, April 1990.

85. Meredith, Jack R. and Mantel, Samuel J., Jr. *Project Management: A Managerial Approach, 2nd edition.* John Wiley & Sons, Inc., New York, New York, 1989.

86. Metzger, Philip W. *Managing a Programming Project, 2nd edition.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1981.

87. Metzger, Philip W. *Managing Programming People: A Personal View.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1987.

88. Miller, Edward and Howden, William E. *Tutorial: Software Testing and Validation Techniques, 2nd edition.* IEEE Computer Society, New York, New York; Los Alamitos, California; Piscataway, New Jersey, 1981.

89. Mills, Harlan D. *Software Productivity.* Little, Brown, and Company, Boston, Massachusetts, 1983.

90. Mills, Harlan D., Linger, Richard C., and Hevner, Alan R. *Principles of Information Systems Analysis and Design.* Academic Press, Orlando, Florida, 1986.

91. Musa, John D., Iannino, Anthony, and Okumoto, Kazuhira. *Software Reliability: Measurement, Prediction, Application.* McGraw-Hill, New York, New York, 1990.

92. Myers, Glenford J. *The Art of Software Testing.* John Wiley & Sons, Inc., New York, New York, 1979.

93. Mynatt, Barbee Teasley. *Software Engineering with Student Project Guidance.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1990.

94. Northrop, Linda M. "Success with the Project-Intensive Model for an Undergraduate Software Engineering Course". *SIGCSE Bulletin 21*, 1 (February 1989), 151-155.

95. Orr, Ken. *Structured Requirements Definition.* K. Orr, Topeka, Kansas, 1981.

96. Ould, Martyn A. and Unwin, Charles, editors. *Testing in Software Development.* Cambridge University Press on behalf of the British Computer Society, Cambridge [Cambridgeshire], England; New York, New York, 1986.

97. Page-Jones, Meilir. *Practical Project Management: Restoring Quality to DP Projects and Systems.* Dorset House Publishing, New York, New York, 1985.

98. Page-Jones, Meilir. *The Practical Guide to Structured Systems Design, 2nd edition.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1988.

99. Parikh, Girish and Zvegintzov, Nicholas, compilers and editors. *Tutorial on Software Maintenance.* IEEE Computer Society, Silver Spring, Maryland, 1983.

100. Peters, Lawrence J. *Software Design: Methods and Techniques.* Yourdon Press, New York, New York, 1981.

101. Peters, Thomas J. and Waterman, Robert H., Jr. *In Search of Excellence: Lessons from America's Best-Run Companies, 1st edition.* Harper & Row, New York, New York, 1982.

102. Pfleeger, Shari Lawrence. *Software Engineering: The Production of Quality Software.* Macmillan; Collier Macmillan, New York, New York; London, England, 1987.

103. Powers, Michael J., Adams, David Robert, and Mills, Harlan D. *Computer Information Systems Development: Analysis and Design.* South-Western Publishing Company, Cincinnati, Ohio, 1984.

104. Pressman, Roger S. *Software Engineering: A Practitioner's Approach, 2nd edition.* McGraw-Hill, New York, New York, 1987.

105. Pressman, Roger S. *Software Engineering: A Beginner's Guide.* McGraw-Hill, New York, New York, 1988.

106. Purnam, Lawrence H. *Tutorial: Software Cost Estimating and Life-Cycle Control Microform: Getting the Software Numbers.* IEEE Computer Society, Los Alamitos, California, 1980.

107. Radice, Ronald A. and Phillips, Richard W. *Software Engineering: An Industrial* Approach. Prentice-Hall International, Englewood Cliffs, New Jersey, 1988.

108. Reifer, Donald J., editor. *Tutorial: Software Management, 3rd edition.* IEEE Computer Society, New York, New York; Los Angeles, California, 1986.

109. Sage, Andrew P., editor. *Systems Engineering: Methodology and Applications.* IEEE Computer Society, New York, New York, 1977.

110. Sanders, Sidney L. "Teaching Load and the Quality of Education". *SIGCSE Bulletin 21*, 4 (December 1989), 27-30.

111. Semprevivo, Philip C. *Systems Analysis - Definition, Process, and Design, 2nd* edition. Science Research Associates, Chicago, Illinois, 1982.

112. Senn, James A. *Analysis and Design of Information Systems.* McGraw-Hill, New York, New York, 1984.

113. Shaw, Mary, editor. *The Carnegie-Mellon Curriculum for Undergraduate Computer Science.* Springer-Verlag, New York, New York, 1985.

114. Shaw, Mary. "Prospects for an Engineering Discipline of Software". *IEEE Software 7*, 6 (November 1990), 15-24.

115. Shneiderman, Ben. *Software Psychology: Human Factors in Computer and Information Systems.* Winthrop Publishers, Cambridge, Massachusetts, 1980.

116. Shooman, Martin L. *Software Engineering: Design, Reliability, and Management.* McGraw-Hill, New York, New York, 1983.

117. Simpson, Henry and Casey, Steven M. *Developing Effective User Documentation.* McGraw-Hill, New York, New York, 1988.

118. Sommerville, Ian. *Software Engineering, 3rd edition.* Addison-Wesley Publishing Company, Wokingham, England; Reading, Massachusetts, 1989.

119. Steward, Donald V. *Software Engineering with Systems Analysis and Design.* Brooks/Cole Publishing Company, Monterey, California, 1987.

120. Teague, Lavette C., Jr. and Pidgeon, Christopher W. *Structured Analysis Methods for Computer Information Systems.* Science Research Associates, Chicago, Illinois, 1985.

121. Thayer, Richard H., compiler. *Tutorial: Software Engineering Project Management.* IEEE Computer Society Press, Washington, DC, 1988.

122. Thayer, Richard H. and Endres, Leo A. "Software Engineering Project Laboratory: The Bridge Between University and Industry". In Gibbs, Norman E. and Fairley, Richard E., Ed., *Software Engineering Education: The Educational Needs of the Software Community*, Springer-Verlag, New York, New York, 1987, pp. 263-291.

123. Tomayko, James E. *Teaching a Project-Intensive Introduction to Software Engineering*. Tech. Rept. CMU/SEI-87-TR-20, DTIC: ADA200603, Software Engineering Institute, Carnegie Mellon University, March 1987.

124. Tomayko, James E. *Materials to Support Teaching a Project-Intensive Introduction to Software Engineering.* Software Engineering Institute, Carnegie Mellon University, 1991. CMU/SEI-91-EM-6.

125. Turner, Ray. *Software Engineering Methodology.* Reston Publishing Company, Reston, Virginia, 1984.

126. Vick, Charles Ralph and Ramamoorthy, Chittoor V. *Handbook of Software* Engineering. Van Nostrand Reinhold Company, New York, New York, 1984.

127. Wallace, Robert H., Stockenberg, John E., and Charette, Robert N. *A Unified Methodology for Developing Systems.* Intertext Publications: McGraw-Hill, New York, New York, 1987.

128. Ward, Paul T. and Mellor, Stephen J. *Structured Development for Real-time Systems.* Yourdon Press, New York, New York, 1986.

129. Wasserman, Anthony I. and Freeman, Peter, editors. *Software Engineering Education: Needs and Objectives, Proceedings of an Interface Workshop.* Springer-Verlag, New York, New York, 1976.

130. Weinberg, Gerald M. *The Psychology of Computer Programming.* Van Nostrand Reinhold, New York, New York, 1971.

131. Wetherbe, James C. *Systems Analysis and Design: Traditional, Structured, and Advanced Concepts and Techniques, 2nd edition.* West Publishing Company, St. Paul, Minnesota, 1984.

132. Whitten, Jeffrey L., Bentley, Lonnie D., and Barlow, Victor M. *Systems Analysis and Design Methods, 2nd edition.* Irwin, Homewood, Illinois, 1989.

133. Wiener, Richard and Sincovec, Richard. *Software Engineering with Modula-2 and Ada.* John Wiley & Sons, Inc., New York, New York, 1984.

134. Yourdon, Edward N., editor. *Classics in Software Engineering.* Yourdon Press, New York, New York, 1979.

135. Yourdon, Edward N., editor. *Writings of the Revolution: Selected Readings on Software Engineering.* Yourdon Press, New York, New York, 1982.

136. Yourdon, Edward N. *Modern Structured Analysis.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1989.

137. Yourdon, Edward N. and Constantine, Larry L. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1979.

Appendix 1: Survey Details

This appendix includes summary material on software engineering courses reported in the Software Engineering Institute's 1990 survey [84]. It gives summary statistics on the courses, analyzes the textbooks reported in that survey, and lists the courses represented.

As noted in Section 5, the reporting from surveyed schools was not uniform. In order to concentrate on software engineering courses, we deleted from our analysis courses whose titles or textbooks suggested that the main emphasis was in areas other than software engineering. We eliminated courses that appeared to be primarily about such topics as:

- Information systems, database management systems
- Programming-in-the-small, programming skills, data structures
- Formal methods, specifications
- User interface design, human factors, graphics
- Artificial intelligence
- Compilers, operating systems
- Networking, communications
- Real-time systems
- Security, privacy
- Procurement and contracting
- Technical writing, communication

Table 3 summarizes some basic information about the remaining 472 courses, which do focus on software engineering. Because graduate courses are often unofficially open to undergraduates, we have included them in the overall summary. Because of incomplete data, the totals are not consistent.

As noted in Section 5, only 126 of the 165 institutions included in the survey provide courses explicitly open to undergraduates. Of those 205 courses, 51 (about 25%) are not necessarily offered every year. Most of the undergraduate courses require prerequisites, suggesting that they can synthesize lower level software development skills. In contrast, nearly a quarter of the graduate courses do not have prerequisites. Some of the prerequisite function might be served by the admissions process of the graduate program, but some of the graduate courses appear to be "re-tread" or conversion courses for software developers with original education in another discipline.

	Undergraduate	Both Grad. & Undergrad.	Graduate	Total
Number				
courses	144	61	263	472
schools	100	42	116	165
Prerequisites?				
yes	138	59	193	391
no	6	2	60	68
Status				
required	74	7	99	181
elective	60	41	144	245
both	3	7	11	21
other	3	3	4	10
Frequency				
biennial	3	9	25	37
once a year	75	30	154	260
once a term	33	7	27	67
alternate terms	4	2	8	14
on demand	4	4	14	22
other	24	7	22	53
Number of Years Taught				
average	4.6	4.7	3.9	4.2
Textbooks				
# distinct texts	44	26	91	116
total # texts assigned	176	72	266	515
mean # uses each text	4.0	2.8	2.9	4.4

Table 3: Statistics on Software Engineering Courses

Of the 472 courses examined, 135 do not assign textbooks, 224 use one text, 84 use two texts, and 29 use three or more. Undergraduate and graduate courses show a slightly different pattern of textbook preferences; these are compared in Table 4, which lists texts in order of popularity. The most notable result of this comparison is the heavy reliance on selected readings in graduate courses but their near-absence in undergraduate courses. Pressman's *Practitioner's Approach* [104] dominates all categories, usually followed by Fairley [47] and Sommerville [118]. Pressman's *Beginner's Guide* [105] and Gane's *Structured System Analysis* [54] are predominantly undergraduate texts, whereas Shooman [116] is predominantly a graduate text. The popularity of Booch's *Software Engineering with Ada* [21] is somewhat surprising in light of the fact that it is essentially an Ada programming text that emphasizes object-oriented development (abstract data types).

Undergraduate	Both Grad & Undergrad	Graduate	Total
Pressman(Pract) [104]	Pressman(Pract) [104]	Selected readings	Pressman(Pract) [104]
Fairley [47]	Booch(SE/Ada) [21]	Pressman(Pract) [104]	Fairley [47]
Sommerville [118]	Sommerville [118]	Fairley [47]	Sommerville [118]
Manuals	Manuals	Sommerville [118]	Manuals
Booch(SE/Ada) [21]	Lamb [77]	Shooman [116]	Selected readings
Brooks [24]	Fairley [47]	Manuals	Booch(SE/Ada) [21]
DeMarco(StrAnal) [40]	Liskov [80]	Booch(SE/Ada) [21]	Brooks [24]
Page-Jones(Design) [98]	Selected readings	Brooks [24]	Shooman [116]
Pressman(Beginner) [105]	Brooks [24]	Conte [35]	Conte [35]
Yourdon(StrAnal) [136]	DeMillo [43]	Boehm [19]	Yourdon(StrAnal) [136]
Gane(Str Anal) [54]	Hausen [61]	Yourdon(StrAnal) [136]	Liskov [80]
Lamb [77]	Myers [92]	Connor [34]	Myers [92]
Myers [92]	(14-way tie for next slot)	DeMarco(Projects) [42]	Page-Jones(Design) [98]
Wiener [133]		Myers [92]	Lamb [77]
(7-way tie for next slot)		Page-Jones(Design) [98]	Boehm [19]

Table 4: Comparison of Text Selections by Course Level

The textbooks, in order of overall popularity, are shown in Table 5. Note that, as is common with such data, the top 10% of the texts account for about 60% of the usage. Accordingly, we include the title in the table for only the 20% of the texts most often selected, and list the other texts by author only. Full citations are in the bibliography.

Cites	Textbook
60	Pressman, *Software Engineering: A Practitioner's Approach* [104]
46	Fairley, *Software Engineering Concepts* [47]
46	Sommerville, *Software Engineering* [118]
32	manuals on languages and tools
32	selected readings
26	Booch, *Software Engineering with Ada* [21]
20	Brooks, *Mythical Man-Month* [24]
15	Shooman, *Software Engineering: Design, Reliability, and Management* [116]
10	Conte, *Software Engineering Metrics and Models* [35]
10	Yourdon, *Modern Structured Analysis* [136]
9	Liskov, *Abstraction and Specification in Program Development* [80]
9	Myers, *The Art of Software Testing* [92]
9	Page-Jones, *The Practical Guide to Structured Systems Design* [98]
8	Lamb, *Software Engineering: Planning for Change* [77]
7	Boehm, *Software Engineering Economics* [19]
6	Pressman, *Software Engineering: A Beginner's Guide* [105]
6	DeMarco, *Structured Analysis and System Specification* [40]
5	Wiener, *Software Engineering with Modula-2 and Ada* [133]
4	Beizer, *Software Testing Techniques* [12]
4	Connor, *Information System Specification and Design Road Map* [34]
4	DeMarco, *Controlling Software Projects: Management, Measurement, and Estimation* [42]
4	Fisher, *CASE (Computer Aided Software Engineering): Using the Newest Tools* [49]
4	Freeman, *IEEE Tutorial on Software Design Techniques* [53]
4	Gane, *Structured Systems Analysis: Tools and Techniques* [54]

Table 5: Textbooks Cited by All Courses in SEI Survey of SE Courses

Used three times:

Babich [5], DeMarco [41],
Gilbert [60], K. & J. Kendall [71],
Metzger [86],
Pfleeger [102], Yourdon (Classics) [134],
Yourdon (Structured Design) [137]

Used twice:

Aron [3], Barstow [7], Bauer [9],
Beck [10], Bentley (Efficient Programs) [14],
Booch (Components) [22],
Brockmann [23],
Cohen [33], Davis (Str Anal) [37],
DeMillo [43], Freedman [51],
Gehani [55], Gilb [59],
Hausen [61], King [74], Kopetz [75],
Meredith [85], Metzger [87], Miller [88],
Peters [100], Peters [101],
Radice [107], Shneiderman [115],
Teague [120], Thayer [121],
Weinberg [130], Whitten [132]

Used once:

Abbott [1], Arthur [4],
Backhouse [6], Basili [8],
Beizer (Test & QA) [11],
Bell [13], Bentley (Pearls) [15],
Bergland [16],
Bersoff [17], Birrell [18],
Bryan (Config Mgt) [26], Bryan (Product Assurance) [27],
Buckley [28], Budde [29],
Chandrasekaran [30],
Charette [31], Chow [32],
Darnell [36], Davis (Requirements) [38],
Deutsch (V&V) [44],
Deutsch (Quality) [45],
Eliason [46], Freeman (Reuse) [52],
Gilb (Metrics) [58], Hetzel [62],
Humphrey [65], Hunke [66],
Jones (SW Dev) [67], Jones (Productivity) [68],
P. Kendall [70]
Kernighan [72], Kezsbom [73],
Kowal [76], Lewis [79], Londeix [81],
Marca [82], Martin [83], Mills [90],
Musa [91], Mynatt [93],
Orr [95], Ould [96],
Page-Jones (Proj Mgt) [97], Parikh [99],
Powers [103],
Putnam [106], Reifer [108], Sage [109],
Semprevivo [111], Senn [112],
Simpson [117],
Steward [119], Turner [125], Vick [126],
Wallace [127], Ward [128], Wetherbe [131]

Table 5: Textbooks Cited, continued

The table on the following pages lists the 205 undergraduate courses that remained after we eliminated courses in the subjects list, as described in Section 5. The table includes courses designated in the SEI survey as open to undergraduates only and courses designated as open to both graduates and undergraduates. We realize that undergraduates are often admitted to graduate courses, but those courses were not designed with undergraduates in mind. Thus the list presented here contains courses *intended* for undergraduates.

Key to Table:

School	Name of college or university
ST	State
Course	Name of course
Level	Level of offering u: undergraduate b: both graduate and undergraduate
Prer	Prerequisites p: course has at least one prerequisite n: no prerequisites x: no information supplied
Stat	Status in requirements r: required e: elective b: both o: other x: no information supplied
Freq	Frequency of offering b: biennial y: once a year t: once a term a: alternate terms d: on demand o: other x: no information supplied
Yrs	Number of years the course has been taught
#texts	Number of textbooks used in the course
next 7 columns	Most popular texts; mark in a column indicates that the text was used in the course. Note that column headings are staggered to save space. Numbers in the first of these columns refer to Pressman, those in the second column refer to Sommerville, etc.
other text	Reference to any other text used in course

Table 6: Software Engineering Course Survey

.

.

.

Table of Contents

List of Figures

List of Tables