Technical Report
CMU/SEI-91-TR-002
ESD-TR-91-002

1991 SEI Report on Graduate Software Engineering Education

Gary Ford

April 1991

Technical Report
CMU/SEI-91-TR-002
ESD-TR-91-002
April 1991

1991 SEI Report on Graduate Software Engineering Education

Gary Ford
Software Engineering Curriculum Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Table of Contents

1991 SEI Report on Graduate Software Engineering Education

Abstract: This report on graduate software engineering education presents a variety of information for university educators interested in establishing a software engineering program. This includes a model curriculum, an annotated bibliography of software engineering textbooks, and descriptions of major software engineering research journals. It also includes detailed descriptions of the the SEI Academic Series of videotape courses, which constitute an example of an implementation of the model curriculum. Twenty-three university graduate programs in software engineering are surveyed. Software engineering textbooks and research journals are also surveyed.

1. Introduction

An ongoing activity of the SEI Education Program is the development and support of a model graduate curriculum in software engineering. In such a rapidly changing discipline, it is important that the curriculum be reevaluated and revised frequently to reflect the state of the art. This report describes our recent efforts toward that end.

Section 2 of this report presents our current recommendations for the content of a Master of Software Engineering degree program. An implementation of these recommendations, the SEI Academic Series, is described in Section 3. For comparison, the graduate software engineering programs of more than 20 universities are surveyed in Section 4.

In response to questions frequently asked of the SEI Education Program, this report also includes two other surveys. Section 5 presents an annotated bibliography of general software engineering textbooks, including comments from professors who have used them. Section 6 lists the major software engineering journals.

Background material is presented in the appendices. Appendices 1 and 2 are taken from [Ford87]; they present, respectively, an organizational structure for discussing software engineering curriculum content and a summary of Bloom's taxonomy of educational objectives. Appendix 3 provides short descriptions of SEI publications that support graduate education, and Appendix 4 describes the history of, and acknowledges the numerous contributors to, the recommendations in this report.

2. A Model Curriculum for a Master of Software Engineering Degree

The academic community distinguishes two master's level technical degrees. The Master of Science in *Discipline* is a research-oriented degree, and often leads to doctoral study. The Master of *Discipline* is a terminal professional degree intended for a practitioner who will be able to rapidly assume a position of substantial responsibility in an organization. The former degree often requires a thesis, while the latter requires a project or practicum as a demonstration of the level of knowledge acquired. The Master of Business Administration (MBA) degree is perhaps the most widely recognized example of a terminal professional degree.

The SEI was chartered partly in response to the perceived need for a greatly increased number of highly skilled software engineers. It is our belief that the SEI can best address this need by encouraging and helping academic institutions to offer a Master of Software Engineering (MSE) degree, or a program with similar content under another title.

In this section we present our current recommendations for a model MSE curriculum. The first subsection (2.0) summarizes the changes since last year's recommendations. The curriculum is then described in seven parts (subsections 2.2-2.8): program objectives, prerequisites, core curriculum content, core topic index, curriculum design for six core courses, the project experience component, and electives. These are followed by short discussions of pedagogical concerns (2.9) and the overall structure of the curriculum (2.10). These recommendations continue to evolve, and we expect to publish updated versions annually.

2.1. Summary of Changes Since 1989

In general, the curriculum recommendations in this report are the same as those in our previous report [Ardis89]. Two additions to the core content are notable.

First, statistical testing concepts and techniques have been added to the unit on software testing. We believe that experiences such as those in [Musa90] indicate both the importance and increased maturity of these techniques. Including these topics in the curriculum will help students gain an appreciation for an engineering approach to testing, rather than an ad hoc, heuristic approach.

Second, a new content unit has been added: Professional Issues. This unit includes legal considerations and professional ethics for software engineers. A number of interesting issues can be discussed with students during the presentation of these topics, including the growing concerns regarding software safety, data security, network security (such as the 1988 Internet worm incident), software piracy, software warranties and licenses, and copyright and patent law applied to programs, algorithms, or user interfaces.

The designs of the six core courses presented in our previous report were developed over a period of several weeks in 1988. Improving those designs would depend in large part on the experiences gained by those who teach the courses. Several universities are now teaching courses based on the designs, but it is too soon to examine their experiences systematically. Therefore, in this report we have neither reprinted the original course designs nor presented improved ones. Instead, we have presented (in Section 3) an example implementation of the designs: the SEI Academic Series. Five of the six courses have been taught, some more than once. Thus they represent the best available example of continued development of the original core course designs.

2.2. Objectives

The goal of the MSE degree program is to produce a software engineer who can rapidly assume a position of substantial responsibility within an organization. To achieve this goal, we propose a curriculum designed to give the student a body of knowledge that includes balanced coverage of the software engineering process activities, their aspects, and the products produced (see Appendix 1 for definitions of the terms *activity*, *aspect*, and *product* as used here), along with sufficient experience to bridge the gap between undergraduate programming and professional software engineering.

Specific educational objectives are summarized below; they appear in greater detail in the descriptions of individual curriculum units in Section 3.3. We describe them using a taxonomy adapted from [Bloom56], which has six levels of objectives: knowledge, comprehension, application, analysis, synthesis, and evaluation. (See Appendix 2 for a brief description of this taxonomy.)

Knowledge: In addition to knowledge about all the material described in the subsequent paragraphs, students should be aware of the existence of models, representations, methods, and tools other than those they learn to use in their own studies. Students should be aware that there is always more to learn, and that they will encounter more in their professional careers, whatever they may have learned in school.

Comprehension: The students should understand:

- the software engineering process, both in the sense of abstract models and in the various instances of the process as practiced in industry
- the activities and aspects of the process
- the issues (sometimes called the *software crisis*) that are motivating the growth and evolution of the software engineering discipline

- the differences between academic or personal programming and software engineering; in particular, students should understand that software engineering involves the production of software systems under the constraints of control and management activities (See Appendix 1 for definitions of these activities)
- a reasonable set of principles, models, representations, methods, and tools
- the role of analysis and evaluation in software engineering
- the existing design paradigms for well-understood systems
- the reasons for and the content of appropriate standards
- the fundamental economic, legal, and ethical issues of software engineering

Application: The students should be able to:

- apply fundamental principles in the performance of the various activities
- apply appropriate formal methods to achieve results
- use appropriate tools covering all activities of the software process
- collect appropriate data for project management purposes, and for analysis and evaluation of both the process and the product
- execute a plan, such as a test plan, quality assurance plan, or configuration management plan; this includes the performance of several kinds of software tests
- apply documentation standards in the production of software engineering documents

Analysis: The students should be able to:

- participate in technical reviews and inspections of various software work products, including documents, plans, designs, and code
- analyze the needs of customers

Synthesis: The students should be able to:

- perform the activities leading to various software work products, including requirements specifications, designs, code, and documentation
- develop plans, such as project plans, quality assurance plans, test plans, and configuration management plans
- design data for and structures of software tests
- prepare oral presentations, and plan and lead software technical reviews and inspections

Evaluation: The students should be able to:

- evaluate software work products for conformance to standards
- use appropriate qualitative and quantitative measures in evaluation of software work products, as in the evaluation of requirements specifications for consistency and completeness, or the measurement of performance

- perform verification and validation of software; these activities should consider all system requirements, not just functional and performance requirements
- apply and validate predictive models, such as those for software reliability or project cost estimation
- evaluate new technologies and tools to determine which are applicable to their own work

The word *appropriate* occurs several times in the objectives above. The software engineering discipline is new and changing, and there is not a consensus on the best set of representations, methods, or tools to use. Each implementation of the MSE curriculum must be structured to match the goals and resources of the school and its students. In subsequent reports, the SEI will offer recommendations on the most promising methods and technologies for many of the software engineering activities.

2.3. Prerequisites

Although an undergraduate degree in computer science is the “obvious” prerequisite for the MSE degree, we cannot adopt such a simplistic approach to defining essential prerequisites. We do not want to exclude experienced practitioners who do not have such a degree but still wish to pursue the MSE degree. Furthermore, students with bachelor’s degrees in computer science from different schools, or from the same school but five years apart, are likely to have substantially different knowledge. Thus the prerequisites for the MSE degree must be defined carefully, and must be enforceable and enforced.

The primary prerequisite, therefore, is substantial knowledge of programming-in-the-small. This includes a working knowledge of at least one modern, high-level language (for example, Pascal, Modula-2, Ada) and at least one assembly language. Also important is a knowledge of fundamental concepts of programming, including control and data structures, modularity, data abstraction and information hiding, and language implementations (runtime environments, procedure linkage, and memory management). Students should also be familiar with the “tools of the trade,” meaning a user’s knowledge (not a designer’s knowledge) of computer organization and architecture, operating systems, and typical software tools (such as an editor, assembler, compiler, and linking loader). A basic knowledge of formal methods and models (and their application) is also essential, including analysis of algorithms and the fundamentals of computability, automata, and formal languages. Most or all of this material is likely to be found in the first three years of an undergraduate computer science degree program.

Knowledge of one or more other major areas of computer science is highly desirable, but not absolutely necessary. Examples are: functional and declarative languages, numerical methods, database systems, compiler construction, computer graphics, or artificial intelligence. This material is usually found in senior-level electives in a computer science degree program. Some schools may choose to allow advanced computer science courses as electives in the MSE program. Knowledge of major applications areas in the sciences and engineering may also be useful.

The mathematics prerequisites are those commonly required in an undergraduate computer science degree: discrete mathematics and some calculus. Some software engineering topics may require additional mathematical prerequisites, such as probability and statistics. A student planning a career in a particular application area may want additional mathematics, such as linear algebra or differential equations, but these are not essential prerequisites for any of the mainstream software engineering courses.

Enforcing the prerequisites can be difficult. A lesson may be learned from experience with master's degree programs in computer science. In the 1960s and 1970s, these programs often served almost exclusively as retraining programs for students with undergraduate degrees in other fields (notably mathematics and engineering) rather than as advanced degree programs for students who already had an undergraduate computer science degree. In several schools, undergraduate computer science majors were not eligible for the master's program because they had already taken all or nearly all of the courses as undergraduates.

These programs existed because there was a clearly visible need for more programmers and computer scientists, and the applicants for these programs did not want a second bachelor's degree. There were not enough applicants who already had a computer science degree to permit enforcement of substantial prerequisites.

For the proposed MSE program to achieve its goals, it must take students a great distance beyond the undergraduate computer science degree. This, in turn, requires that students entering the program have approximately that level of knowledge. Because of the widely varying backgrounds of potential students, their level of knowledge is very difficult to assess. Standardized examinations, such as the Graduate Record Examination in Computer Science, provide only part of the solution.

We recommend that schools wishing to establish the MSE program consider instituting a *leveling* or *immigration* course to help establish prerequisite knowledge. Such a course rarely fits into the normal school calendar. Rather, it is an intensive two to four week course that is scheduled just before or just after the start of the school year. (However, Texas Christian University has tried a full-semester leveling course; see [Comer86]). Students receive up to 20 hours a week of lectures summarizing all of the prerequisite material. The value of this course is not that the students become proficient in all the material, but that they become aware of deficiencies in their own preparation. Self-study in parallel with the first semester's courses can often remove most of these deficiencies.

Another important part of the immigration course is the introduction of the school's computing facilities, especially the available software tools. Ten to 20 hours each week can be devoted to demonstrations and practice sessions. Because proficiency with tools can greatly increase the productivity of the students in later courses, the time spent in the immigration course can be of enormous value.

Finally, the immigration course can be used to help motivate the study of software engineering. The faculty, and sometimes the students themselves, can present some of their own or others' experiences that led to improved understanding of some of the significant problems of software engineering.

Another kind of prerequisite has been adopted by some MSE programs (including the University of St. Thomas, Seattle University, and Texas Christian University). All require the student to have at least one year of professional experience as a software developer. This requirement has the benefit of giving the students increased motivation for studying software engineering: professional experience exposes them to the problems of developing systems that are much larger than those seen in the university, and makes them aware of economic and technical constraints on the software development process. On the negative side, schools cannot control the quality of that experience, and students may acquire bad habits that must be unlearned.

We have not found the arguments for an experience prerequisite sufficiently compelling to recommend it for all MSE programs. Other engineering disciplines have successful master's level programs, and even undergraduate programs, without such a prerequisite. Most graduate professional degrees in other disciplines do not require it.

As a discipline grows and evolves, it is a common phenomenon in education for new material to be taught in courses that are simply added onto an existing curriculum. Over time, the new material is assimilated into the curriculum in a process called *curricular compression*. Obsolete material is taken out of the curriculum, but much of the compression is accomplished by reorganization of material to get the most value in the given amount of time.

In a rapidly growing and changing discipline, new material is added faster than curricular compression can accommodate it. In some engineering disciplines, the problem is acute. There is a growing sentiment that the educational requirement for an entry-level position in engineering should be a master's degree or a five-year undergraduate degree [NRC85]. This is especially true for a computer science/ software engineering career.

If this level of education is needed for a meaningful entry-level position, then we question the value of sending students out with a bachelor's degree, hoping they will return sometime later for a software engineering degree. The professional experience achieved during that time will not necessarily be significant. Also, the percentage of students intending to return to school who actually do return declines rapidly as time since graduation increases. Therefore, we believe that an MSE curriculum structured to follow immediately after a good undergraduate curriculum offers the best chance of achieving the goals of rapid increases in the quality and quantity of software engineers. Of course, such a program does not preclude admission of students with professional experience.

We do recognize that work experience can be valuable. The experience component of the MSE curriculum, which is discussed later in this report, might be structured to include actual work experience. It may be that the overall educational experience is significantly enhanced if the work component is a coordinated part of the program rather than an interlude between undergraduate and graduate studies.

We also recognize that we must provide motivation for many of the activities in the software engineering process. We see a great need to raise the level of awareness on the part of both students and educators of the differences between undergraduate programming and professional software engineering. The SEI Education Program is working at the undergraduate level to help accomplish this.

2.4. Core Curriculum Content

Software engineering is a broad and diverse discipline. To facilitate discussions of the content of software engineering curricula, we have found it helpful to develop an organizational structure for the discipline; this is presented in Appendix 1. A brief look at this structure is sufficient to conclude that all of software engineering cannot be covered in any curriculum. Selecting a subset of that content appropriate for a particular program and student population is the primary task of a curriculum designer.

We use a broad view of software engineering when choosing the content of the curriculum, and we include several topics that are not part of a typical engineering curriculum. In this respect, we agree with this statement of the National Research Council about engineering curricula [NRC85]:

> …[T]o make the transition from high school graduate to a competent practicing engineer requires more than just the acquisition of technical skills and knowledge. It also requires a complex set of communication, group-interaction, management, and work-orientation skills.
>
> ... For example, education for management of the engineering function (as distinct from MBA-style management) is notably lacking in most curricula. Essential nontechnical skills such as written and oral communication, planning, and technical project management (including management of the individual’s own work and career) are not sufficiently emphasized.

On the other hand, we have narrowed the curriculum by concentrating almost exclusively on *software* engineering (but including some aspects of *systems* engineering) and omitting applications area knowledge. The two major reasons for this are pragmatic: first, the body of knowledge known as software engineering is sufficiently large to require all the available time in a typical master's degree program (and then some); and second, students cannot study all of the applications areas in which they might eventually work. We believe that students at the graduate level should have acquired the skills for self-education that will enable them to acquire needed knowledge in an application area.

More important, however, is our strong belief that the variety of applications areas and the level of sophistication of hardware and software systems in each of those areas mandate a development *team* with a substantial range of knowledge and skills. Some members of the team must understand the capabilities of hardware and software components of a system in order to do the highest level specification, while other members must have the skills to design and develop individual components. Software engineers will have responsibility for software components just as electrical, mechanical, or aeronautical engineers, for example, will have responsibility for the hardware components. Scientists, including computer scientists, will also be needed on development teams; and all the scientists and engineers must be able to work together toward a common goal.

The core content of the MSE curriculum is described in *units*, each covering a major topic area, rather than in courses. There are three reasons for this. First, not every topic area contains enough material for a typical university course. Second, combining units into courses can be accomplished in different ways for different organizations. Third, this structure more easily allows each unit to evolve to reflect the changes in software engineering knowledge and practice while maintaining the stability of the overall curriculum structure.

Because of strong relationships among topics and subtopics, we were unable to find a consensus on an appropriate order of topics. We do, however, recommend a top-down approach that begins with focus on the software engineering process; this overall view is needed to put the individual activities in context. Software management and control activities are presented next, followed by the development activities and product view topics.

Social and ethical issues are also important to the education and development of a professional software engineer. Examples are privacy, data security, and software safety. We do not recommend a course on these issues, but rather encourage instructors to find opportunities to discuss them in appropriate contexts in all courses and to set an example for students.

The curriculum topics are described below in units of unspecified size. Nearly all have a software engineering activity as the focus. For each, we provide a short description of the subtopics to be covered, the aspects of the activity that are most important, and the educational objectives of the unit. (See Appendix 1 for definitions of the terms *activity* and *aspect* as they are used here.)

1. The Software Engineering Process

Topics The software engineering process and software products. All of the software engineering activities. The concepts of software process model and software product life cycle model.

Aspects All aspects, as appropriate for the various activities.

Objectives Knowledge of activities and aspects. Some comprehension of the issues, especially the distinctions among the various classes of activities. The students should begin to understand the substantial differences between the programming they have done in an undergraduate program and software engineering as it is practiced professionally.

2. Software Evolution

Topics The concept of a software product life cycle. The various forms of a software product, from initial conception through development and operation to retirement. Controlling activities and disciplines to support evolution. Planned and unplanned events that affect software evolution. The role of changing technology.

Aspects Models of software evolution, including development life cycle models such as the waterfall, iterative enhancement, phased development, and spiral models.

Objectives Knowledge and comprehension of the models. Knowledge and comprehension of the controlling activities.

3. Software Generation

Topics Various methods of software generation, including designing and coding from scratch, use of program or application generators and very high level languages, use of reusable components (such as mathematical procedure libraries, packages designed specifically for reuse, Ada generic program units, and program concatenation, as with pipes). Role of prototyping. Factors affecting choice of a software generation method. Effects of generation method on other software development activities, such as testing and maintenance.

Aspects Models of software generation. Representations for software generation, including design and implementation languages, very high level languages, and application generators. Tools to support generation methods, including application generators.

Objectives Knowledge and comprehension of the various methods of software generation. Ability to apply each method when supported by appropriate tools. Ability to evaluate methods and choose the appropriate ones for each project.

4. Software Maintenance

Topics	Maintenance as a part of software evolution. Reasons for maintenance. Kinds of maintenance (perfective, adaptive, corrective). Comparison of development activities during initial product development and during maintenance. Controlling activities and disciplines that affect maintenance. Designing for maintainability. Techniques for maintenance, including program reading and reverse engineering.
Aspects	Models of maintenance. Current methods.
Objectives	Knowledge and comprehension of the issues of software maintenance and current maintenance practice. Ability to apply basic maintenance techniques.

5. Technical Communication

Topics	Fundamentals of technical communication. Oral and written communication. Preparing oral presentations and supporting materials. Software project documentation of all kinds.
Aspects	Principles of communication. Document preparation tools. Standards for presentations and documents.
Objectives	Knowledge of fundamentals of technical communication and of software documentation. Application of fundamentals to oral and written communications. Ability to analyze, synthesize, and evaluate technical communications.

6. Software Configuration Management

Topics	Concepts of configuration management. Its role in controlling software evolution. Maintaining product integrity. Change control and version control. Organizational structures for configuration management.
Aspects	Fundamental principles. Tools. Documentation, including configuration management plans.
Objectives	Knowledge and comprehension of the issues. Ability to apply the knowledge to develop a configuration management plan and to use appropriate tools.

7. Software Quality Issues

Topics Definitions of quality. Factors affecting software quality. Planning for quality. Quality concerns in each phase of a software life cycle, with special emphasis on the specification of the pervasive system attributes. Quality measurement and standards. Software correctness assessment principles and methods. The role of formal verification and the role of testing. Concepts of reliability and reliability modeling. Fundamental issues of software security.

Aspects Assessment of software quality, including identifying appropriate measurements and metrics. Tools to help perform measurement. Correctness assessment methods, including testing and formal verification. Formal models of program verification.

Objectives Knowledge and comprehension of software quality issues and correctness methods. Knowledge and comprehension of concepts of software reliability modeling and software security. Ability to apply proof of correctness methods.

8. Software Quality Assurance

Topics Software quality assurance as a controlling discipline. Organizational structures for quality assurance. Independent verification and validation teams. Test and evaluation teams. Software technical reviews. Software quality assurance plans.

Aspects Current industrial practice for quality assurance. Documents including quality assurance plans, inspection reports, audits, and validation test reports.

Objectives Knowledge and comprehension of quality assurance planning. Ability to analyze and synthesize quality assurance plans. Ability to perform technical reviews. Knowledge and comprehension of the fundamentals of program verification and its role in quality assurance. Ability to apply concepts of quality assurance as part of a quality assurance team.

9. Software Project Organizational and Management Issues

Topics Project planning: choice of process model, project scheduling and milestones. Staffing: development team organizations, quality assurance teams. Resource allocation.

Aspects Fundamental concepts and principles. Scheduling representations and tools. Project documents.

Objectives Knowledge and comprehension of concepts and issues. It is not expected that a student, after studying this material, will be ready to manage a software project immediately.

10. Software Project Economics

Topics Factors that affect cost. Cost estimation, cost/benefit analysis, risk analysis for software projects.

Aspects Models of cost estimation. Current techniques and tools for cost estimation.

Objectives Knowledge and comprehension of models and techniques. Ability to apply the knowledge to tool use.

11. Software Operational Issues

Topics Organizational issues related to the use of a software system in an organization. Training, system installation, system transition, operation, retirement. User documentation.

Aspects User documentation and training materials.

Objectives Knowledge and comprehension of the major issues.

12. Requirements Analysis

Topics The process of interacting with the customer to determine system requirements. Defining software requirements. Identifying functional, performance, and other requirements: the pervasive system requirements. Techniques to identify requirements, including prototyping, modeling, and simulation.

Aspects Principles and models of requirements. Techniques of requirement identification. Tools to support these techniques, if available. Assessing requirements. Communicating with the customer.

Objectives Knowledge and comprehension of the concepts of requirements analysis and the different classes of requirements. Knowledge of requirements analysis techniques. Ability to apply techniques and analyze and synthesize requirements for simple systems.

13. Specification

Topics Objectives of the specification process. Form, content, and users of specifications documents. Specifying functional, performance, reliability, and other requirements of systems. Formal models and representations of specifications. Specification standards.

Aspects Formal models and representations. Specification techniques and tools that support them, if available. Assessment of a specification for attributes such as consistency and completeness. Specification documents.

Objectives Knowledge and comprehension of the fundamental concepts of specification. Knowledge of specification models, representations, and techniques, and the ability to apply or use one or more. Ability to analyze and synthesize a specification document for a simple system.

14. System Design

Topics The role of system design and software design. How design fits into a life cycle. Software as a component of a system. Hardware versus software tradeoffs for system performance and flexibility. Subsystem definition and design. Design of high-level interfaces, both hardware to software and software to software.

Aspects System modeling techniques and representations. Methods for system design, including object-oriented design, and tools to support those methods. Iterative design techniques. Performance prediction.

Objectives Comprehension of the issues in system design, with emphasis on engineering tradeoffs. Ability to use appropriate system design models, methods, and tools, including those for specifying interfaces. Ability to analyze and synthesize small systems.

15. Software Design

Topics Principles of design, including abstraction and information hiding, modularity, reuse, prototyping. Levels of design. Design representations. Design practices and techniques. Examples of design paradigms for well-understood systems.

Aspects Principles of software design. One or more design notations or languages. One or more widely used design methods and supporting tools, if available. Assessment of the quality of a design. Design documentation.

Objectives Knowledge and comprehension of one or more design representations, design methods, and supporting tools, if available. Ability to analyze and synthesize designs for software systems. Ability to apply methods and tools as part of a design team.

16. Software Implementation

Topics Relationship of design and implementation. Features of modern procedural languages related to design principles. Implementation issues, including reusable components and application generators. Concepts of programming support environment.

Aspects One or more modern implementation languages and supporting tools. Assessment of implementations: coding standards and metrics.

Objectives Ability to analyze, synthesize, and evaluate the implementation of small systems.

17. Software Testing

Topics The role of testing and its relationship to quality assurance. The nature and limitations of testing. Levels of testing: unit, integration, acceptance, etc. Statistical testing methods. Detailed study of testing at the unit level. Formal models of testing. Test planning. Black box and white box testing. Building testing environments. Test case generation. Test result analysis.

Aspects Testing principles and models. Tools to support specific kinds of tests. Assessment of testing; testing standards. Test documentation.

Objectives Knowledge and comprehension of the role and limitations of testing. Ability to apply test tools and techniques. Ability to analyze test plans and test results. Ability to synthesize a test plan.

18. System Integration

Topics Testing at the software system level. Integration of software and hardware components of a system. Uses of simulation for missing hardware components. Strategies for gradual integration and testing.

Aspects Methods and supporting tools for system testing and system integration. Assessment of test results and diagnosing system faults. Documentation: integration plans, test results.

Objectives Comprehension of the issues and techniques of system integration. Ability to apply the techniques to do system integration and testing. Ability to develop system test and integration plans. Ability to interpret test results and diagnose system faults.

19. Embedded Real-Time Systems

Topics Characteristics of embedded real-time systems. Existence of hard timing requirements. Concurrency in systems; representing concurrency in requirements specifications, designs, and code. Issues related to complex interfaces between devices and between software and devices. Criticality of embedded systems and issues of robustness, reliability, and fault tolerance. Input and output considerations, including unusual data representations required by devices. Issues related to the cognizance of time. Issues related to the inability to test systems adequately.

Objectives Comprehension of the significant problems in the analysis, design, and construction of embedded real-time systems. Ability to produce small systems that involve interrupt handling, low-level input and output, concurrency, and hard timing requirements, preferably in a high-level language.

20. Human Interfaces

Topics Software engineering factors: applying design techniques to human interface problems, including concepts of device independence and virtual terminals. Human factors: definition and effects of screen clutter, assumptions about the class of users of a system, robustness and handling of operator input errors, uses of color in displays.

Objectives Comprehension of the major issues. Ability to apply design techniques to produce good human interfaces. Ability to design and conduct experiments with interfaces, to analyze the results and use them to improve the design.

21. Professional Issues

Topics Issues of professionalism in software engineering. Ethics. Legal issues including intellectual property rights, warranties, and liability.

Objectives Comprehension of basic concepts and issues of professional behavior. Comprehension of major ethical issues. Knowledge of the major legal issues.

2.5. Core Curriculum Topic Index

The core curriculum topics presented in the previous section are organized into 21 content units, each containing many related topics. Because there are many more topics than units, it is not always obvious in which unit a topic may be found. Therefore, we have included here an alphabetical index of topics. The number after each topic is the content unit in which that topic may be found.

tools
- configuration management 6
- cost estimation 10
- design 15
- documentation 5
- implementation 16
- requirements identification 12
- scheduling 9
- software generation 3
- software quality measurement 7
- specification 13
- system design 14
- test 17

tradeoffs between hardware and software 14
training 11
training materials 11
user documentation 11
validation test reports 8
very high level language 3
virtual terminals 20
warranties 21
waterfall model 2
writing, technical 5

2.6. Curriculum Design

The 21 content units described in Section 2.4 can be considered a *specification* for the core curriculum; a *design*, then, is a description of the courses that present the core material in an appropriate, coherent way. At the 1988 SEI Curriculum Design Workshop (described in [Ardis89]), participants developed six semester-length courses as one such design. Forming the basis for the model MSE curriculum, these courses are:

Software Systems Engineering
Specification of Software Systems
Principles and Applications of Software Design
Software Generation and Maintenance
Software Verification and Validation
Software Project Management

Detailed course descriptions may be found in a previous SEI report [Ardis89]. We have not redesigned these courses since that report was published. Instead of repeating those descriptions here, we have chosen to present an actual implementation of five of those courses (see Section 3 of this report). The sixth course is being implemented and taught during the fall 1990 semester, and it will be described in a later report.

2.7. Project Experience Component

In addition to coursework covering the units described in Section 2.4, the curriculum should incorporate significant software engineering experience representing at least 30% of the student's work. Universities have tried a number of approaches to give students this experience; examples are summarized in Figure 2.1.

School	**Approach**	**Description**
Seattle University, Monmouth College, Texas Christian University	Capstone project course	Students do a software development project after completion of most coursework
University of Southern California	Continuing project	Students participate in a project that continues from year to year (the *Software Factory*), building and enhancing software engineering tools and environments
Arizona State University	Multiple course coordinated project	A single project is carried through four courses (on software analysis, design, testing, and maintenance); students may take the courses in any order
University of Stirling	Cooperative program with industry	After one year of study, students spend six months in industry on a professionally managed software project, followed by a semester of project or thesis work based in part on the work experience
Imperial College	Commercial software company	Students participate in projects of a commercial software company that has been established by the college in cooperation with local companies
Carnegie Mellon University	Design studio	Students work on a project under the direction of an experienced software designer, in a relationship similar to that of an apprentice and master

Figure 2.1. Approaches to the experience component

One form of experience is a cooperative program with industry, which has been common in undergraduate engineering curricula for many years. The University of Stirling uses this form in their Master of Science in Software Engineering program [Budgen86]. Students enter the program in the fall semester of a four-semester program. Between the first and second semesters, they spend two or three weeks in industry to learn about that company. They return to the company in July for a six-month stay, during which time they participate in a professionally managed project. The fourth semester is devoted to a thesis or project report, based in part on their industrial experience.

Imperial College of Science and Technology has a similar industry experience as part of a four-year program leading to a Master of Engineering degree [Lehman86]. For this purpose, the college has set up Imperial Software Technology, Ltd. (IST) in partnership with the National Westminster Bank PLC, The Plessey Company PLC, and PA International. IST is an independent, technically and commercially successful company that provides software technology products and services.

The more common form of experience, however, is one or more project courses as part of the curriculum. Two forms are common: a project course as a capstone following all the lecture courses, and a project that is integrated with one or more of the lecture courses.

The Wang Institute of Graduate Studies (before it closed in 1987), Texas Christian University, and Seattle University have each offered a graduate software engineering degree for several years, and the University of St. Thomas has had a program for six years. Each school incorporates a capstone project course into its curriculum. The Wang Institute often chose projects related to software tools that could be useful to future students. TCU takes the professional backgrounds of its students into consideration when choosing projects. Seattle sometimes solicits real projects from outside the university. The University of St. Thomas allows students to work on projects for their employers, if the projects are outside their normal work assignments.

It is worth noting that none of these institutions mention software maintenance in their project course descriptions. Yet, educators and practitioners alike have long recognized that maintenance requires the majority of resources in most large software systems. The lack of coverage of maintenance in software engineering curricula may be attributed to several factors. First, there does not appear to be a coherent, teachable body of knowledge on software maintenance. Second, current thinking on improving the maintenance process is primarily based on improving the development process; this includes the capturing of development information for maintenance purposes. Finally, giving students maintenance experience requires that there already exists a significant software system with appropriate documentation and change requests, the preparation of which requires more time and effort than an individual instructor can devote to course preparation. (The SEI has published some materials to address this final problem [Engle89].)

The University of Southern California has built an infrastructure for student projects that continue beyond the boundaries of semesters and groups of students. The *System Factory Project* [Scacchi86] has created an experimental organizational environment for developing large software systems that allows students to encounter many of the problems associated with professional software engineering and to begin to find effective solutions to the problems. To date, more than 250 graduate students have worked on the project and have developed a large collection of software tools.

The University of Toronto has added the element of software economics to its project course [Horning76, Wortman86]. The *Software Hut* (a small software house) approach requires student teams to build modules of a larger system, to try to sell their module to other teams (in competition with teams that have developed the same module), to evaluate and buy other modules to complete the system, and to make changes in purchased modules. At the end of the course, systems are “sold” to a “customer” at prices based on the system quality (as determined by the instructor’s letter grade for the system). The instructor reports that this course has a very different character from previous project courses. The students’ attempts to maximize their profits gave the course the flavor of a game and helped motivate students to use many techniques for increasing software quality.

Arizona State University has built the project experience into a sequence of courses, combining lectures with practice [Collofello82]. The courses–Software Analysis (requirements and specifications), Software Design, Software Testing, and Software Maintenance–were offered in sequence so that a single project could be continued through all four. However, the students could take the courses in any order; and although many students did take them in the normal (waterfall model) order, the turnover in enrollment from one semester to the next gave a realistic experience.

Carnegie Mellon University has recently initiated an MSE degree program based in part on the SEI curriculum described in this report. This program was originally designed to include a year-long *design studio* approach to the project experience component, in which students work closely with faculty on software development. This approach is similar to the *master-apprentice* model common in the education of engineers and craftsmen in the 19th century, and it is specifically modeled after the studio courses now common in architecture and fine arts programs. After a prototype offering of the studio in the spring and summer of 1990, the course was modified. Students now register for one hour of studio during the fall semester, when they develop project requirements and specifications in conjunction with the software systems engineering course; and they register for one hour in the spring semester, when they develop a project plan as part of the project management course. They then execute the plan during a summer-fall design studio course sequence. This new approach is expected to make better use of the students’ time and to integrate the project work with the core courses.

We do not believe that there is only one *correct* way to provide software engineering experience. It can be argued that experience is the basis for understanding the abstractions of processes that make up formal methods and that allow reasoning about processes. Therefore, we should give the students experience first, with some guidance, and then show them that the formalisms are abstractions of what they have been doing. It can also be argued that we should teach “theory” and formalisms first, and then let the students try them in capstone project courses.

No matter what form the experience component takes, it should provide as broad an experience as possible. It is especially important for the students to experience, if not perform, the control activities and management activities (as defined in Appendix 1). Without these, the project can be little more than advanced programming.

2.8. Electives

Electives may comprise 20% to 40% of a curriculum. Although software engineering is a young discipline, it is already sufficiently broad that students can choose specializations (such as project management, systems engineering, or real-time systems); there is no “one size fits all” MSE curriculum. The electives provide the opportunity for that specialization.

In addition, there is a rather strong perception among industrial software engineers that domain knowledge for their particular industry is essential to the development of effective software systems. Therefore, we also suggest that an MSE curriculum permit students to choose electives from the advanced courses in various application domains. Software engineers with a basic knowledge of avionics, radar systems, or robotics, for example, are likely to be in great demand. Furthermore, there is increasing evidence that better software project management can significantly influence the cost of software, so electives in management topics are appropriate.

To summarize, there are five recommended categories of electives:

1. Software engineering subjects, such as software development environments
2. Computer science topics, such as database systems or expert systems
3. Systems engineering topics, especially topics at the boundary between hardware and software
4. Application domain topics
5. Engineering management topics

2.9. Pedagogical Considerations

Software engineering is difficult to teach for a variety of reasons. First, it is a relatively new and rapidly changing discipline, and it has aspects of an art and a craft as well as a science and an engineering discipline. As a result, educators must develop a variety of teaching techniques and materials in order to provide effective education.

Secondly, psychologists distinguish *declarative* knowledge and *procedural* knowledge [Norman88]. The former is easy to write down and easy to teach; the latter is nearly impossible to write down and difficult to teach. Procedural knowledge is largely subconscious, and it is best taught by demonstration and best learned through practice. It is because many of the processes of software engineering depend on procedural knowledge that we recommend such a significant amount of project experience (see Section 2.7).

Another aspect of experience that can be built into the curriculum involves “tricks of the trade.” Software engineers, during the informal apprenticeship of their first several years in the profession, are likely to be exposed to a large number of recurring problems for which there are accepted solutions. These problems and solutions will vary considerably from one application domain to another, but all software engineers seem to accumulate them in their “bags of tricks.”

We believe that students would receive some of the benefits of their “apprenticeship” period while still in school if these problems and solutions were included in the curriculum. For this reason, we have included large course segments titled “Paradigms” in the specification and design courses (see the original designs of these courses in [Ardis89] and the course descriptions in Sections 3.3 and 3.4).

The principal definition of the word *paradigm* is “EXAMPLE, PATTERN; *esp* : an outstandingly clear or typical example or archetype” [Webster83]. The word *archetype* is defined in the same source as “the original pattern or model of which all things of the same type are representations or copies : PROTOTYPE; *also* : a perfect example.” We believe that these definitions capture the notion of a widely accepted or demonstrably superior solution to a recurring problem.

Unfortunately, there is no ready source of appropriate paradigms. The paradigms sections of the specification and design courses represent the current instructors’ best thinking on appropriate paradigms. We hope to continue to identify the most important recurring problems in many application domains and to incorporate the best paradigms into these courses.

2.10. The Structure of the MSE Curriculum

A typical master’s degree curriculum requires 30 to 36 semester hours[†] of credit. The courses described in Section 3.4 require three hours each, totaling 18 semester hours. This allows time for the project experience component and for some electives.

Because of the wide range of choices for electives, students can be well served by creative course design. For example, several small units of material (roughly one semester hour each) might be prepared by several different instructors. Three of these could then be offered sequentially in one semester under the title “Topics in Software Engineering,” with different units offered in different semesters.

[†]Note for readers not familiar with United States universities: A *semester hour* represents one contact hour (usually lecture) and two to three hours of outside work by the student per week for a semester of about fifteen weeks. A *course* covers a single subject area of a discipline; the class typically meets three hours per week, and the student earns three semester hours of credit. A graduate student with teaching or research responsibilities might take three courses (nine semester hours) each semester; a student without such duties might take five courses.

Figure 2.2 shows the structure of a curriculum based on the six recommended core courses. This structure reflects the familiar *spiral approach* to education, in which material is presented several times in increasing depth. This approach is essential for a discipline such as software engineering, with many complex interrelationships among topics; no simple linear ordering of the material is possible.

Students learn the basics of computer science and programming-in-the-small in the undergraduate curriculum. The six core courses build on these basics by adding depth, formal methods, and the programming-in-the-large concepts associated with systems engineering and control and management activities. The electives and the project experience component provide further depth and an opportunity for specialization.

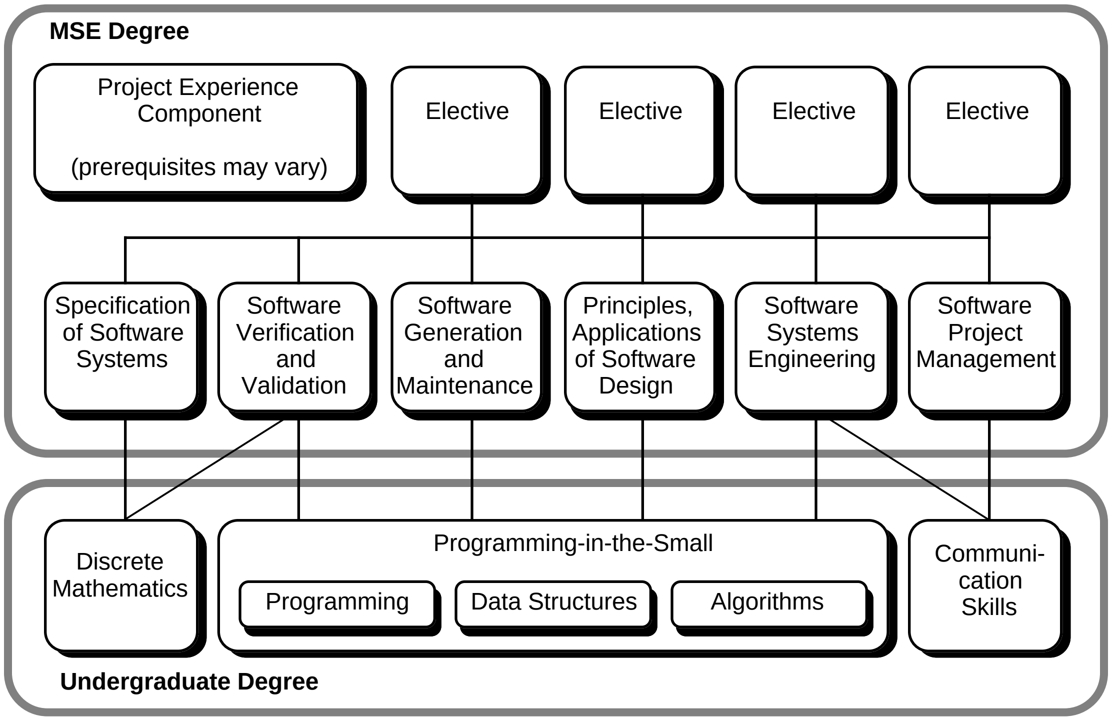

MSE Degree
Project Experience Component
(prerequisites may vary)
Elective
Elective
Elective
Elective
Specification of Software Systems
Software Verification and Validation
Software Generation and Maintenance
Principles, Applications of Software Design
Software Systems Engineering
Software Project Management
Discrete Mathematics
Programming-in-the-Small
Programming
Data Structures
Algorithms
Communi-cation Skills
Undergraduate Degree

Figure 2.2. MSE curriculum structure

3. The SEI Academic Series Videotape Courses

In 1988, the SEI Education Program began teaching a series of graduate-level software engineering courses. Although Carnegie Mellon University students were able to register for these courses, the primary purpose was to make the courses available to other universities on videotape. These courses came to be known as the *Academic Series* courses; other SEI videotape courses are the *Continuing Education Series* and the *Technology Series*.

The instructors for the courses used the course designs developed at the 1988 SEI Curriculum Design Workshop (see [Ardis89]). As might be expected, the process of implementing the courses uncovered some rough areas in the designs. Some of the courses have now been offered two or three times, and thus they represent a substantial refinement of the original designs. Some remaining problems are that a few of the individual topics in the core curriculum content (see section 2.4 of this report) are missing, and a few other topics show up in more than one course. Continued development of these courses is expected in conjunction with Carnegie Mellon's recently established MSE program.

In this chapter, we describe in detail five of the six core courses. (The sixth course was implemented and taught during the fall 1990 semester, and it will be described in a later report.) Each description includes:

- short discussions of student prerequisites, course objectives, and the instructor's philosophy
- a syllabus giving the titles of lectures
- a summary of each lecture (based on the instructor's classroom transparencies), with reading assignments for the students

In some cases, the lecture summaries are quite short; this was necessary when the lecture was primarily a detailed presentation of a particular example, when the material covered was highly graphical or otherwise did not lend itself to a written summary, and when detailed instructor's notes were not available.

Special acknowledgement goes to the three SEI staff members who implemented and taught these courses. Mark Ardis developed the courses Specification of Software Systems and Software Verification and Validation. Robert Firth developed the courses Principles and Applications of Software Design and Software Creation and Maintenance. James Tomayko developed Software Project Management.

Our goal in presenting this material is to help other instructors design courses or individual lectures in these areas. We are also investigating the feasibility of making individual videotape lectures available to universities, so this material can serve as a "catalog" of those tapes. For further information on the availability of these tapes, please contact the leader of the Software Engineering Curriculum Project at the SEI.

3.1. Software Systems Engineering

A prototype version of this course was taught for the first time in the SEI Academic Series videotape courses during the fall semester 1990. A detailed summary of the lectures is not yet available. The information below is reprinted from the original course design in [Ardis89].

Students' Prerequisites

Students should have knowledge of software life cycle models, computer architectures, and basic statistics.

Objectives

After completing this course, students should comprehend the alternative techniques used to specify and design systems of software and hardware components. They should be able to find the data and create a requirements document and to develop a system specification. They should understand the concepts of simulation, prototyping, and modeling. They should know what is needed to prepare a system for delivery to the user and what makes a system usable.

Philosophy of the Course

This course exposes students to the development of software systems at the very highest level. It introduces the system aspect of development and the related tradeoffs required when software and hardware are developed together, especially with respect to user interfaces. It exposes students to requirements analysis and techniques for developing a system from those requirements. System integration and transition into use are also covered.

Syllabus

Wks Topics and Subtopics *(Objective)*

1 Introduction *(Knowledge)*

Students should see the "big picture" in this part of the course. The emphasis should be on how software is only one component of a larger system.

Overview of topics

1 System Specification *(Comprehension)*

Contents

Standards

Global issues such as safety, reliability

2 System Design *(Comprehension)*

Simulation

Queuing theory

Tradeoffs

Methods (levels, object-oriented, function-oriented)

3 Interfaces (Comprehension)

Both human interfaces and interfaces to hardware devices should be included. These areas require different skills but are logically combined here to emphasize the notion of encapsulation of software within larger systems.

Human factors

Guidelines

Experiments

Devices

1 System Integration *(Comprehension)*

Students should learn how to perform integration of entire systems, not just software.

Simulation of missing components

System build

5 Requirements Analysis *(Synthesis)*

This is the largest part of the course. Students should learn the interpersonal skills as well as the technical skills necessary to elicit requirements from users. Expression and analysis of requirements are often performed with CASE tools.

Objectives

Interview skills

Needs and task analysis

Prototypes

SADT, RSL (and other specific methods)

1 Operations Requirements *(Comprehension)*

Students should understand and know how to satisfy the other operations requirements of systems, such as training and documentation.

Training

Online help

User documentation

Pedagogical Concerns

Case studies should be available as assigned readings. A requirements analysis project should be assigned to students, with topics in the lectures sequenced to match the project schedule. A user interface prototype project should be assigned, including an exercise in user documentation. The students should give a presentation on their requirements study. An instructor of this course should have experience in requirements analysis and system design.

Comments

We had a great deal of difficulty naming this course. Much of the work that students will perform as exercises and projects deals with requirements analysis. On the other hand, this course attempts to place software in perspective with other elements of systems. The theme of the course is not just requirements analysis, but total systems engineering. We noted that universities often have courses titled “systems engineering” that cover the same topics from an electrical engineering perspective.

An important goal of this course is that students achieve an understanding of the role of software engineering within the larger context of systems engineering. They should understand, for example, that while ensuring that a software system satisfies its specification is a *software* problem, getting the right specification is a *systems* problem. If software does not give the right system behavior, it must be determined whether the software fails to meet the specification or whether the specification does not define the right system behavior. These distinctions are critical as students leave the academic world, where the entire system is often a personal computer, and enter the “real world” of embedded systems.

3.2. Specification of Software Systems

Students' Prerequisites

Students must have a reasonable level of knowledge of:
- set theory
- functions and relations
- predicate calculus
- axioms
- finite-state machines (transition mapping, nondeterminism)

Objectives

- Knowledge of major models of specification (sequential and concurrent systems)
- Knowledge of existing standards and practices
- Mastery of one method of describing functional behavior of simple sequential systems
- Appreciation of advantages and disadvantages of formality

Philosophy of the Course

A comparative survey approach serves to emphasize similarities and significant differences between languages and methods. For example, students specify the same system with three or four different languages. Whenever possible, common paradigms are translated into their language-specific idioms.

Students are expected to develop skills in reading formal specifications, reasoning about formal descriptions, modelling "standard" paradigms, and translating models into formal notations. The exercises and exams provide practice and feedback on these skills, especially on small problems.

Syllabus

The syllabus assumes 27 class meetings, including midterm and final examinations. Each meeting is planned to include approximately 55 to 60 minutes of lecture and 20 minutes of class discussion.

1. Introduction
2. Readers and Writers
3. Standards
4. Algebraic Model
5. Larch
6. State-Machine
7. ASLAN
8. Abstract Models
9. VDM
10. Z
11. Industrial Use
12. Midterm Review
13. Midterm Examination
14. PAISLey 1
15. PAISLey 2
16. Concurrency Paradigms
17. Petri Nets: Concepts
18. Petri Nets: Modeling
19. Communicating Sequential Processes (CSP) 1
20. Communicating Sequential Processes (CSP) 2
21. Synchronous Calculus of Communicating Systems (SCCS) 1
22. Synchronous Calculus of Communicating Systems (SCCS) 2
23. Temporal Logic
24. Statecharts
25. Final Review
26. Final Examination

Summaries of Lectures

1. Introduction

Reading: Cohen86, Chapter 1

The course will address these topics: what specifications are, the role of specification in software development, the process of creating specifications, the process of using specifications, and the advantages and disadvantages of various notations for specifications.

Current industrial practice includes specifications that are formal or informal, abstract or detailed, and standardized or ad hoc. Specifications may be written by the client or by the developer, by the application specialist or the specification specialist. They are read by all these people.

There are several models for specifications. Sequential models and languages include algebraic (exemplified by Larch), state-machine (ASLAN), abstract models (VDM, Z), and operational (Paisley). Concurrency models and languages include communicating processes (Paisley, CSP, CCS), state-machines (Petri nets, Statecharts), and predicate logic (temporal logic).

2. Readers and Writers

Reading: Parnas86

The *IEEE Standard Glossary of Software Engineering Terminology* defines *specification* as "A document that prescribes, in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristics of a system or system component. See also design specification, formal specification, functional specification, interface specification, performance specification, requirements specification."

Two kinds of specifications may be distinguished: customer-oriented specifications (C-Specs) express requirements from the customer's point of view; developer-oriented specifications (D-Specs) express requirements from the developer's point of view. Some C-Spec issues include why the user wants the system, how the user intends to use the system, and the expertise of people using the system. Some D-Spec issues include the behavioral attributes of the system, a functional description, and performance characteristics.

A specification may be viewed as a contract between the customer and the developer. It should be unambiguous, complete, verifiable, consistent, modifiable, traceable, and usable. Some common deficiencies are poor organization, boring prose, confusing and inconsistent terminology, and "myopia" (working at too low a level).

Some types of specifications are function (what the system must do), interface (how the system will be used by other components, including people), and performance (how the system must perform, such as speed and accuracy). The organization of the specification may be reference, historical, or ad hoc; the tone may be narrative or declarative.

Many people have a role in the specification process; these include users, customers, requirements analysts, designers, verification and validation staff, and maintainers.

3. Standards

Reading: IEEE84, Jones89, Duce88, ANSI85

There are two important relationships between specifications and standards: standards for writing specifications and the specification of a standard. Two examples of standards for writing specifications are the *IEEE Guide to Software Requirements Specifications* and the *Department of Defense Military Standard 2167A*.

The IEEE standard includes a discussion of specification issues and suggests an outline for specification documents. Four different organizations are provided for the section "Specific Requirements." The first has separate subsections for functional specifications, interface specifications, performance specifications, and constraints. This approach has the advantage of avoiding duplication. The second combines functional with interface specifications, but has separate subsections for performance and constraints. This approach emphasizes how to use the system. The third organization combines performance and constraints with each functional requirement, and then combines performance and constraints with each interface specification; this approach emphasizes the interface. The fourth organization combines all the issues, addressing interface, performance, and constraint issues individually for each functional requirements. This approach emphasizes the independence of functional requirements from one another.

Some of the issues in specification are the relative sizes of the components of a specification, the relative importance of the components, the ways in which the specification will be read, and the ways in which it will be maintained.

Standards are different from systems in that: they describe systems; many systems may satisfy the same standard; it is difficult to create a standard before there are any systems; and a standard is an abstraction of a system. Standards should be specified in a way that emphasizes the important abstractions and allows easy verification of conformance.

[The lecture includes a detailed example: the GKS (graphics kernel system) standard in its ANSI form (expressed in English) and in VDM form (as described in Duce88).]

4. Algebraic Model

Reading: Cohen86, Chapters 2-4

The main issues are notations and foundations, how to write algebraic specifications, and consistency and completeness. The algebraic view of specification is that all operations are related to one another. Four basic idioms are: f undoes g, f is independent of g, f ignores (part of) g, and f is (sometimes) the same as g.

The foundations include: a set of axioms defines a *congruence relation*—some sequences of operations are equivalent to other sequences; any implementation that is *equivalent up to isomorphism* (operations may be consistently renamed) is considered correct; specified objects (types) may be used in other specifications; a *consistent* set of axioms avoids contradictions of the form "true = false"; a *complete* set of axioms provides enough information to uniquely determine the result of any sequence of operations; those operations that return values in old types are *behavior* functions; those operations that return old values in the new type are *modifier* functions; those operations that return new values in the new type are *constructor* functions.

The components of an algebraic specification are the syntactic part (containing such items as the names of domains or types, the names of operations, and the domains and ranges of operations), and the semantic part (containing the meanings of operations).

Current technology for working with algebraic specifications includes: Larch at MIT (Guttag), CMU (Wing), and DEC SRC (Horning); Hope, Clear at Edinburgh (Burstall); OBJ at UCLA and SRI (Goguen); and Project CIP at Munich (Bauer).

[The lecture includes detailed examples of the algebraic specification of stack, queue, and symbol table data types; the examples illustrate Guttag's *sufficiently complete* method of writing axioms.]

5. Larch

Reading: Guttag85; Roberts88 optional

The Larch language uses a two-tiered approach to specifications. The specification language is based on *traits* defined by axioms and on *operators* for combining traits. The interface language provides links between specification names and program names, plus additional constraints. Programs typically include objects such as types, packages, modules and procedures, and functions.

Traits may be combined in a number of ways. The "assumes" operator refers to components of another trait; "includes" creates a larger trait by inclusion. Other combining operators include generation, partitioning, conversion, exemption, substitution, and inclusion.

Interface languages are a mapping between a programming language and the Larch Shared Language (LSL). For example, such a language mapping Pascal to LSL will map Pascal types to LSL *sorts*, procedures and functions to operators, and abstract data types to traits.

Assertions in Larch include "requires" for preconditions, "modifies" for side effects, and "ensures" for postconditions. All assertions are written in predicate calculus, with references to traits and to built-in predicates (from programming language semantics).

Practical advice for using Larch is to begin by finding a canonical representation—a sequence of operations that can be used to create any value of the type of interest. Add behavior operators to simplify expressions; a condition that naturally arises often in describing behavior should be captured in a behavior operator. Then look for common right-hand sides. The four basic idioms: "nochange" indicates that there is no simpler description; "ignores" indicates that some operations are discarded; "commutes" indicates the order of operations is reversed; "undoes" indicates that the effect of the last operation is undone.

6. State-Machine

Reading: Parnas72

This method describes a data type as a state-machine inside a black box with push buttons. Some of the buttons change the state of the machine and others display information about the state. The objective of the method is to hide as much as possible, while providing access to as much as needed. This includes providing all needed information to the user, and no more; providing all needed information to implementor, and no more; using a formal model, so that consistency and completeness can be decided; and not inventing a new model.

State-changing operations are called O-functions; the description of each should include exceptions (error cases) and new values of V-functions. The information-displaying operations are called V-functions; the description of each includes exceptions and initial values.

The advantages of the state machine method include: they are minimal and do not bias the implementation; the basic idea of a state machine is already familiar to most programmers; and completeness is easy to show. Disadvantages include: it may require hidden functions; and the relationships between functions may be obscure, thus causing difficulty with demonstrating consistency. Some problems with the method include specification of delayed effects and asymmetry (implicit operands).

[The lecture includes detailed examples of the specification of the stack data type and the Pascal file type.]

7. ASLAN

Reading: Auernheimer84, Kemmerer85

ASLAN is a language for writing state-machine specifications. The O- and V-functions of the state machine model can be translated to ASLAN. The O-functions are called *transitions* and the V-functions are declared as *variables* (state components). Initial values are collected together. Error cases for transitions are called *entry conditions* and those for variables are called *invariants*. A specification consists of declarations (types, constants, etc.), requirements (invariants, etc.), and transitions.

Additional features of ASLAN include declarations (abstraction), implied "no changes," and a small programming language for added power of expression.

Correctness conjectures in ASLAN are addressed through *invariants* and *constraints*,which express properties that should be true of the specified objects. ASLAN generates a set of *verification conditions* from these predicates. Axioms may be used to simplify the verification conditions.

To begin the process of writing ASLAN specifications, model the problem domain in terms of a collection of state-machines, identifying types and transitions. Then identify the state components, invariants, and constraints. Write the ASLAN specification, and then run the ASLAN processor to generate the verification conditions. Finally, prove the verification conditions, iterating where necessary.

[The lecture includes a detailed discussion of an ASLAN specification of a stack data type.]

8. Abstract Models

Reading: Cohen86, Chapter 5; Bjørner 78, Bjørner87, Lucas87 for background

The major mathematical concepts behind abstract models include sets, lists, records, maps, and predicates. An abstract syntax is a system of functions for constructing and decomposing composite objects.

[The lecture includes a detailed discussion of the grocer example from Bjørner87.]

9. VDM

VDM (Vienna Development Method) was created at the IBM Vienna laboratory. It is being used in programming language definition for compilers, for database specification, and in office automation applications.

The Meta-IV language includes elementary types (boolean, numbers, etc.), composite types (sets, tuples, etc.), and combinators (states, blocks, etc.). Normal use adheres to naming conventions. VDM specifications include semantic domains (abstract model), invariants (well-foundedness), and functions (operations). A British variant has operations described via pre- and post-conditions, significant components of the "state" identified for each operation, and minor syntax variations.

The method of VDM is based on stepwise refinement (decomposition, reification) and proof obligations: at each step of the refinement a *retrieve* function is defined, its adequacy proved, and commutativity of operations with respect to the function are proved.

[The lecture includes detailed discussions of VDM specifications of the stack and queue data types, the bank example from Jones86, and a reification example.]

10. Z

Reading: Spivey89; Hayes87 and Spivey88 for background

Z (pronounced in the British manner as "zed") is another model-theoretic method, with incrementality via schema calculus. It is founded on set theory and other mathematics (in a library), and on first-order predicate logic. Important features include schema notations, naming conventions, mathematical library, ways of combining schema (conjunction, disjunction, hiding, overriding, composition, and piping).

The Z method is to introduce basic sets; define an abstract state in terms of sets, relations, functions, sequences, etc.; specify the initial state; define the pre- and post-conditions of operations; state and prove theorems; and refine toward the concrete, incurring proof obligations. Current technology to support the method includes text processing tools and consistency checkers; other tools are being developed.

[The lecture includes a detailed discussion of the symbol table example from Hayes87.]

11. Industrial Use

Reading: Delisle89, Hayes85, Nix88, Ruggles88, Woodcock88

Four examples illustrate the use of Z in industry: the IBM CICS project, GKS standardization, the GEC Telecom storage allocator, and the Tektronix oscilloscope description.

The IBM project was maintenance of the Customer Information Control System (CICS). Z was used to reverse engineer old code: the product was over 20 years old, over 500,000 lines of code, had many users and many configurations. Restructuring of the code was necessary before Z could be used; this was done independently of any method. Then the Z specifications were derived from manuals, developers, and existing code. Ultimately about half of CICS was described in Z (230,000 lines of code), and modules were added or rewritten from the Z specifications. The normal IBM development process was used, including design reviews, code inspections, and testing. The normal IBM programming languages were used, plus a guarded command language. It was necessary to retrain the staff in the use of Z; this included existing IBM courses on discrete mathematics and a software engineering workshop, augmented with Z courses. The final results were that more time was spent in design, inspections required less preparation but took longer to conduct, and more problems were found earlier in design and fewer problems found in testing. The product has been shipped and the developers are awaiting customer reaction.

The effort to standardize the graphics kernel system (GKS) via formal descriptions has been under way for many years. The first GKS standard was released in 1985, written in a procedural style. VDM and algebraic methods have been used to describe parts of the standard in a more abstract form; Z is now being tried. Originally VDM was the leading contender for the specification, but it was found to be inadequate with respect to concurrency and modularity. A strong sense of intuition is now developing that the main abstractions of GKS can best be described algebraically, but progress has been hindered by the lack of a standard for VDM and the lack of a consensus on which method to use.

The GEC Telecom project used VDM in the maintenance of the storage allocator for the operating system of digital exchanges. The project involved 12 staff members for 1.5 years, producing 30,000 lines of code. Their process included training all staff in VDM, using reviews but not proofs, using a “scouting” strategy (in which experts looked ahead for trouble spots), and a consensus strategy (the description of state was resolved before moving on to the next stage). GEC’s results included finding that education was difficult, due to the lack of special courses on the variant of VDM being used; the lack of standardization of VDM caused problems for the staff and tools; reworking specifications was at least as hard as writing them (two-thirds of the time was spent reviewing and reworking); changing requirements caused extra work; the difficulty of project tasks was underestimated; newcomers to the project were able to contribute quickly; many errors were discovered early; the resulting design was simpler but similar to the original; and it was easier to implement from the design.

Tektronix undertook a small (two-person) research project to explore the specification of devices. Previous device descriptions had been operational. The project team talked with engineers, tried to describe existing devices, and discussed the specifications with engineers. They found a useful abstraction that clarified the role of hardware and software engineers, and the specifications yielded insight into design tradeoffs for user interfaces, sampling methods, and hardware/software partitioning. Lessons learned included: engineers in industry can understand formal specifications; abstraction was useful in focussing attention on the right problem; and specification was a process rather than a product.

Common themes in all four industry examples were: formal methods are more time-consuming, at least during early stages of the life cycle; traditional development processes may still be used; more errors are found earlier in development; and modularity and standardization are important.

12. Midterm Review

13. Midterm Examination

14. PAISLey 1

Reading: Zave82, Horning73

Operational methods describe a system by a program or set of programs. They may be formal if the programming language has a formal semantics; they may be executable if the programming language has an interpreter or compiler.

PAISLey (process-oriented, applicative, and interpretable specification language) allows writing an executable specification as a step between informal requirements and code. It was designed for real-time and distributed systems and allows specification of performance. Several tools are available, including a parser for finding syntax errors, an interpreter for performing simulations, a cross-referencer, and a consistency checker. It has been used for specification of a submarine lightguide system and a finite-element adaptive research solver.

The process model views a system as a set of asynchronous processes. Each new process cycles through a sequence of states described by a mapping from the old state to the new state. Processes communicate by “exchange functions” that may or may not involve waiting. Processes have internal synchronous concurrency via components of tuples. Exchange functions need to model synchronizing and free-running processes, and self-matching and competing processes.

The data model is based on types constructed from atomic values (such as integers, reals, strings, or symbolic values) and set operations (such as union, cross-product, and enumeration). Values may be either atomic or tuples.

The computational model is based on independently specified processes, with exchange functions (in terms of mappings) for communication. The interpreter selects a mapping to evaluate. Within a process this is done nondeterministically; between processes a first-in, first-out strategy is used for xm-type exchanges. The interpreter maintains a system-wide clock to be used in measuring performance. Concurrent evaluation occurs between processes (except for exchanges) and within tuple evaluation.

To specify performance requirements, timing constraints may be attached to mappings. These may be upper bounds, lower bounds, or constant time. Uniformly distributed random values are used when the evaluation time is not constant.

During interpretation (simulation), timing constraints are checked; violations generate error messages.

15. PAISLey 2

Reading: Zave85; Bruns86 and Zave86 for background

The transformational method of developing a specification requires that a correct specification be developed at each level of abstraction. Each level is refined to move closer to implementation and to optimize inefficient solutions. The most significant issue is deciding which objects to refine at each level.

The JSD (Jackson System Development) method is based on entities that model objects in the real world and actions that model events in the real world. A process network is a set of communicating concurrent processes. Its specification is developed by identifying entities, identifying processes, arranging them in time-order, identifying inputs and outputs of processes, arranging them into a network, refining the network with functions, and refining the data with tuples and sequences.

The PAISLey method develops a specification by identifying entities, identifying processes, describing them with a process model (cycle), identifying inputs and outputs of processes, connecting them with exchange functions, refining processes with functions, and refining data with tuples and sets.

Criteria for decomposing a system into processes include: each process corresponds to a “meaningful activity” of the system; the system is decomposed so that data items modified by a process are related; and each process contains activities with similar cycle times. Criteria for defining process cycles include: there is maximal use of parallelism in process steps; each process step corresponds to a “meaningful cycle” of the activity; process cycles are time-bounded unless there is a good reason for making an exception; bounded iteration is used to traverse homogeneous tuples. Criteria for defining inter-process interactions include: exchange-function types and channels for all interactions are identified so that processes can satisfy timing constraints; exchange-function names are clear; and buffered message passing is not introduced for performance reasons only. An important criterion for deciding the level of abstraction in a specification is to avoid defining sets and mappings if they are well-understood, available from a library, or better specified using another formalism. Criteria to divide specifications into modules include: information-hiding is used to partition the specification into files; abstract data types are identified and placed in separate files; and the specification is partitioned into files so that interesting, executable subsets can be run without manual intervention.

16. Concurrency Paradigms

Reading: Horning73, Andrews83

A process may be defined as a triple, consisting of a state space, a successor function, and initial states. The state space is a set of (variable, value) pairs. The successor function maps states into states, and it may be a deterministic function or a nondeterministic relation. Combinations of processes are said to be disjoint if their state spaces are disjoint, serial if only one process may be active at a time, synchronous if each process takes one step at the same time, and asynchronous if the processes operate at different rates. Interactions between processes are called cooperation if they are anticipated and desired; interference if they are unanticipated or unacceptable; and competition if they are anticipated and acceptable, but undesirable.

Shared variable paradigms of concurrency involve concepts of mutual exclusion, busy waiting, semaphores, conditional critical regions, monitors, and path expressions. Message passing paradigms involve concepts of channels, pipelines, client/server relationships, ports, blocking, and buffering. Higher-level descriptions involve concepts of remote procedure calls, rendezvous, and atomic transactions.

Two major properties of systems are important. *Safety* properties describe something that should not happen or properties that hold over finite sequences of events. *Liveness* properties describe something that must happen or properties that hold over infinite sequences of events; three liveness properties are *termination*, *deadlock*, and *starvation*. The termination property specifies that every process that should terminate does. Deadlock is the condition in which all processes are blocked, waiting for conditions that can only be changed by blocked processes. Starvation is the condition in which some process never gets unblocked, even though the system is not deadlocked.

17. Petri Nets: Concepts

Reading: Peterson77

A *Petri net* is a bipartite directed graph (P, T, E), where P is a set of *places*; T is a set of *transitions*; and E is a set of directed *edges* from a place to a transition or from a transition to a place. A *marking* M of a Petri net is a mapping from P to the natural numbers; it assigns a number of *tokens* to places. Petri nets are often defined with an *initial marking*. A transition may *fire* if it is *enabled*. It is enabled if each of its input places has at least one token. Firing of a transition removes a token from each input place and produces a token at each output place. The *state space* of a Petri net is the set of all markings (a marking is a state). A marking m' is *immediately reachable* from marking m if the firing of some enabled transition in m yields m'. The *reachable* relation is the reflexive transitive closure of the immediately reachable relation. The *reachability set* is the set of all reachable states.

Important properties of Petri nets include: they allow concurrent execution (firing of transitions); they are nondeterministic; synchronization may be specified; resource allocation may be modeled.

A Petri net is said to be a *safe net* if it has no more than one token at any place. It is *k-bounded* if it has no more than k tokens at any place. It is *bounded* if it has no more than k tokens at any place, but we don't know the value of k. It is *conservative* if the number of tokens in the net remains constant.

The *liveness* problem for Petri nets may be expressed as "are all transitions potentially fireable?" The *reachability problem* is "is state m' reachable from state m?" The *coverability* problem is "given states m and m', does there exist a marking m" reachable from m, such that m" ≥ m'?" A *reachability tree* can be used to solve safeness, boundedness, conservation, and coverability problems. The reachability tree is a finite representation of a potentially infinite structure.

[The lecture includes detailed examples of specification of an elevator system, a producer/consumer problem, and mutual exclusion via Petri nets.]

18. Petri Nets: Modeling

Reading: Agerwala79

[This lecture consists of detailed examples of Petri net modeling methods, including synchronization, shared resources, and semaphores.]

19. Communicating Sequential Processes (CSP) 1

Reading: Hoare78

Fundamental concepts of CSP include simple event sequences, recursively defined sequences, and choice of events. Several improvements to CSP from the definition in Hoare78 have been introduced in Hoare85, including: change of symbol to reuse libraries, ports and channels, no assumption of automatic termination, traces-language of events, and proof methods to use when showing that two levels of specification are consistent.

[The lecture includes detailed examples of a vending machine, a producer/consumer problem, and the dining philosophers problem.]

20. Communicating Sequential Processes (CSP) 2

Reading: Kallstrom88 and Wayman87 for background

Processes often interact. Processes are described by events. The set of all events of a process is called its *alphabet*. If concurrent processes share alphabets, then they must simultaneously participate in the events that they share.

There are algebraic laws that form a calculus for CSP. When trying to show that a property (theorem) is true for a given specification, we can manipulate CSP expressions with these laws. Specifications can be written in CSP. Other important concepts are satisfaction, choice, communication, and livelock.

[The lecture includes detailed examples of interaction of processes using the vending machine and dining philosophers problems; an example of CSP specification for the vending machine problem; and a discussion of the bakery algorithm.]

21. Synchronous Calculus of Communicating Systems (SCCS) 1

Reading: Cohen86, Chapter 6; Milner83 and Diaz89 for background

CCS (calculus of communication systems) was invented first; afterward, Milner realized that a generalization yielded explicit modeling of time, hence SCCS. A surprise was the discovery that CCS was a subcalculus of SCCS. It emphasizes interfaces between agents (the action set). Time is modeled discretely. Semantics are operationally based. It has as few operators as possible.

The action of delay has no externally observable effect (except to use time). So adding a delay does not change the behavior of a process. Similarly, the internal communication between subprocesses yields only an externally visible delay. So, let delay be represented by the identity element of the calculus (multiplication by 1), and let internal communication be shown by simultaneous execution of inverses.

Two machines are *observationally equivalent* if they allow the same experiments to be performed and they produce the same externally visible behavior for any experiment.

[The lecture includes a detailed comparison of examples of CSP and SCCS.]

22. Synchronous Calculus of Communicating Systems (SCCS) 2

Synchronous specifications are important, but many systems are asynchronous; communications protocols and airline reservation systems are examples. Therefore it is worth examining CCS; a knowledge of SCCS is assumed.

Fundamental definitions include *synchrony*: synchronistic occurrence, arrangement or treatment; *synchronous*: happening, existing or arising at precisely the same time; *asynchrony*: absence or lack of concurrence in time. Basic concepts include event sequences; unbuffered communication on channels known as *ports* (although the calculus is asynchronous, communication synchronizes the agents); fairness; relabeling; restriction; summation and composition over a set; strong equivalence; observational equivalence.

CCS may be derived from SCCS. A convincing argument is to derive each CCS operator from SCCS operators. The key step is the introduction of a new SCCS operator, *delay*. We conclude that rather than use two different calculi for synchrony and asynchrony, the synchronous calculus alone is sufficient. It is also simpler than the asynchronous calculus.

CCS has been used for protocol specification, and it forms the basis of LOTOS (an ISO standard).

[The lecture includes detailed examples of a 1-place message buffer, a buffer with a persistent state, a nondeterministic choice, mutual recursion, a fair buffer, and a 2-place buffer.]

23. Temporal Logic

Reading: Barringer87, Lamport83, Pneuli85, Rescher71, and Wood89 for background

Temporal logic extends predicate logic through the introduction of temporal operators, such as *eventually*, *henceforth*, *until*, *next*, and *previous*. It can be used to reason about the behavior of systems through time.

An example application is the specification of the behavior of a simple elevator system, including safety and liveness properties. The example demonstrates that temporal logic can be the system specification; a finite state machine is the model of operation; underspecification is easy to accomplish, difficult to determine, and relatively easy to correct; the system is allowed any behavior not explicitly forbidden; and that an implementation can be derived from either the specification or from the model. Verification of the specification can be handcrafted or tool supported, both with difficulty. The handcrafted verification is too tedious to be done without error, and the existing tools are often too limited for any but the smallest of specifications.

[The lecture includes a detailed discussion of the temporal logic specification of the elevator system.]

24. Statecharts

Reading: Harel87, Harel88

[This lecture provides a detailed introduction to the statechart notation and concepts.]

25. Final Review

26. Final Examination

Bibliography

Agerwala79	Agerwala, Tilak. "Putting Petri Nets to Work." *Computer 12*, 12 (Dec. 1979), 85-94.
Andrews83	Andrews, Gregory R. and Schneider, Fred B. "Concepts and Notations for Concurrent Programming." *ACM Computing Surveys 15*, 1 (Mar. 1983), 3-43.
ANSI85	*American National Standard for Information Systems — Computer Graphics — Graphical Kernel System (GKS) Functional Description*. ANSI X3.124-1985, American National Standards Institute, 1985.
Auernheimer84	Auernheimer, Brent and Kemmerer, Richard A. *ASLAN User's Manual*. Tech. Rep. TRCS84-10, Department of Computer Science, University of California, Santa Barbara, Santa Barbara, Calif., Aug. 1984.
Barringer87	Barringer, H. "Up and Down the Temporal Way." *Computer J. 30*, 2 (Apr. 1987), 134-148.
Bjørner78	*The Vienna Development Method: The Meta-Language*. Dines Bjørner; Cliff Jones, eds. Berlin: Springer-Verlag, 1978. Vol. 61, Lecture Notes in Computer Science.
Bjørner87	*VDM '87: VDM—A Formal Method at Work. VDM-Europe Symposium 1987*. D. Bjørner; C. B. Jones; M. Mac an Airchinnigh; E. J. Neuhold, eds. Berlin: Springer-Verlag, 1987. Vol. 252, Lecture Notes in Computer Science.
Bruns86	Bruns, Glenn R. *Technology Assessment: PAISLEY*. Tech. Rep. MCC TR STP-296-86, MCC, Austin, Texas, Sept. 1986.
Cohen86	Cohen, B., Harwood, W. T., and Jackson, M. I. *The Specification of Complex Systems*. Reading, Mass.: Addison-Wesley, 1986. This book is now out of print.
Delisle89	Delisle, Norman and Garlan, David. "Formally Specifying Electronic Instruments." *Proc. Fifth International Workshop on Specification and Design*. IEEE Computer Society, May 1989, 242-248. Also published as *ACM Software Engineering Notes 14* (3).

Diaz89	Diaz, M. and Vissers, C. "SEDOS: Designing Open Distributed Systems." *IEEE Software 6*, 6 (Nov. 1989), 24-33.
Duce88	Duce, D. A., Fielding, E. V. C., and Marshall, L. S. "Formal Specification of a Small Example Based on GKS." *ACM Transactions on Graphics 7*, 7 (July 1988), 180-197.
Gehani86	Gehani, N. and McGettrick, A. *Software Specification Techniques*. Reading, Mass.: Addison-Wesley, 1986.
Guttag78	Guttag, John V., Horowitz, Ellis, and Musser, David R. "Abstract Data Types and Software Validation." *Comm. ACM 21*, 12 (Dec. 1978), 1048-1064.
Guttag85	Guttag, John V., Horning, James J., and Wing, Jeanette M. "The Larch Family of Specification Languages." *IEEE Software 2*, 5 (Sept. 1985), 24-36.
Harel87	Harel, David. "Statecharts: A Visual Formalism for Complex Systems." *Science of Computer Programming 8* (1987), 231-274.
Harel88	Harel, D., Lachover, H., Naamad, A, A. Pnueli, Politi, M., Sherman, R., and Shtul-Trauring, A. "Statemate: A Working Environment for the Development of Complex Reactive Systems." *10th International Conference on Software Engineering*. IEEE, 1988, 396-406.
Hayes85	Hayes, I. J. "Applying Formal Specification to Software Development in Industry." *IEEE Trans. Software Engineering SE-11*, 2 (Feb. 1985), 169-178.
Hayes87	Hayes, I. J. *Specification Case Studies*. Englewood Cliffs, N. J.: Prentice-Hall International, 1987.
Hoare78	Hoare, C. A. R. "Communicating Sequential Processes." *Comm. ACM 21*, 8 (Aug. 1978), 666-677.
Hoare85	Hoare, C. A. R. *Communicating Sequential Processes*. London: Prentice-Hall International, 1985.
Holzmann91	Holzmann, Gerard J. *Design and Validation of Computer Protocols*. Englewood Cliffs, N. J.: Prentice-Hall, 1991.
Horning73	Horning, J. J. and Randell, B. "Process Structuring." *ACM Computing Surveys 5*, 1 (Mar. 1973), 5-30.
IEEE84	*IEEE Guide to Software Requirements Specifications*. Std. 830-1984, IEEE, 1984.
Jones86	Jones, Cliff. *Systematic Software Development Using VDM*. Englewood Cliffs, N. J.: Prentice-Hall International, 1986.
Jones89	Jones, Peggy. *DASC Requirements Document.* In *Software Maintenance Exercises for a Software Engineering Project Course*, CMU/SEI-89-EM-1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Feb. 1989.

Kallstrom88 Kallstrom, Marta and Thakkar, Shreekant S. “Programming Three Parallel Computers.” *IEEE Software 5*, 1 (Jan. 1988), 11-22.

Kemmerer85 Kemmerer, Richard A. “Testing Formal Specifications to Detect Design Errors.” *IEEE Trans. Software Engineering SE-11*, 1 (Jan. 1985), 32-43.

Lamport83 Lamport, Leslie. “What Good is Temporal Logic.” *Information Processing: Proceedings of the IFIP World Computer Congress*. New York: Elsevier, 1983, 657-668.

Lucas87 Lucas, Peter. “VDM: Origins, Hopes and Achievements.” *Proceedings of VDM-Europe Symposium ’87*, Bjørner, Dines, et al, ed. Berlin: Springer-Verlag, 1987. Vol. 252, Lecture Notes in Computer Science.

Milner83 Milner, R. “Calculi for Synchrony and Asynchrony.” *Theoretical Computer Science 25* (Nov. 1983), 267-310.

Nix88 Nix, C. J. and Collins, B. P. “The Use of Software Engineering, Including the Z Notation, in the Development of CICS.” *Quality Assurance 14*, 3 (Sept. 1988).

Owicki82 Owicki, Susan and Lamport, Leslie. “Proving Liveness Properties of Concurrent Programs.” *ACM Trans. Programming Lang. and Syst. 4*, 3 (July 1982), 455-495.

Parnas72 Parnas, David L. “A Technique for Software Module Specification with Examples.” *Comm. ACM 15*, 5 (May 1972), 330-336. Also in Gehani86, p. 75-88.

Parnas86 Parnas, David L. and Clements, Paul C. “A Rational Design Process: How and Why to Fake It.” *IEEE Trans. Software Engineering SE-12*, 2 (Feb. 1986), 251-257.

Peterson77 Peterson, James L. “Petri Nets.” *ACM Computing Surveys 9*, 3 (Sept. 1977), 223-252.

Pneuli85 Pneuli, A. “Application of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends.” *Current Trends in Concurrency: Overviews and Tutorials.* deBakker, J. W., deRoever, W. P., and Rosenberg, G., eds. New York: Springer-Verlag, 1985, 510-584. Vol. 224, Lecture Notes in Computer Science.

Rescher71 Rescher, Nicholas and Urquart, Alasdair. *Temporal Logic.* New York: Springer-Verlag, 1971.

Roberts88 Roberts, W. T. “A Formal Specification of the QMC Message System.” *Computer Journal 31*, 4 (Aug. 1988), 313-324.

Ruggles88	Ruggles, Clive. "Towards a Formal Definition of GKS and Other Graphics Standards." *VDM '88: VDM — The Way Ahead*, Bloomfield, R., Marshall, L., and Jones, R., eds. New York: Springer-Verlag, 1988, 64-73. Vol. 328, Lecture Notes in Computer Science.
Schwartz81	Schwartz, Richard L. and Melliar-Smith, P. M. "Temporal Logic Specification of Distributed Systems." *Proc. 2nd Intl. Conf. on Distributed Computing Syst.* New York: IEEE, 1981, 446-454.
Spivey88	Spivey, J. M. *The Z Notation: A Reference Manual*. Englewood Cliffs, N. J.: Prentice-Hall, 1988.
Spivey89	Spivey, J. M. "An Introduction to Z and Formal Specifications." *Software Engineering Journal 4*, 1 (Jan. 1989), 40-50.
Wayman87	Wayman, Russell. "OCCAM 2: An Overview from a Software Engineering Perspective." *Microprocessors and Microsystems 11*, 8 (Oct. 1987).
Wood89	Wood, William G. *Temporal Logic Case Study*. Tech. Rep. CMU/SEI-89-TR-24, ADA219019, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 1989.
Woodcock88	Woodcock, J. C. P. and Dickinson, B. "Using VDM with Rely and Guarantee-Conditions." *VDM '88: VDM — The Way Ahead*, Bloomfield, R., Marshall, L., and Jones, R., eds. New York: Springer-Verlag, 1988, 449-455. Vol. 328, Lecture Notes in Computer Science.
Zave82	Zave, Pamela. "An Operational Approach to Requirements Specification for Embedded Systems." *IEEE Trans. Software Engineering SE-8*, 3 (May 1982), 250-269. Also in Gehani86, 131-169.
Zave85	Zave, Pamela. "A Distributed Alternative to Finite-State-Machine Specifications." *ACM Trans. Prog. Lang. and Systems 7*, 1 (Jan. 1985), 10-36.
Zave86	Zave, Pamela and Schell, William. "Salient Features of an Executable Specification Language and Its Environment." *IEEE Trans. Software Engineering SE-12*, 2 (Feb. 1986), 312-325.

3.3. Principles and Applications of Software Design

Students' Prerequisites

- computer science background
- reasonable knowledge of programming
- some experience of team software development
- some exposure to the software development *process*
- industrial experience is an advantage

Objectives

Help the student make the transition from programming-in-the-small to programming-in-the-large:

- appreciate the software development process
- understand the role of design in that process
- become familiar with the major design methods
- be able to categorize and comprehend a new design method

Enable the student to undertake software design:

- select an appropriate method
- apply the method properly and intelligently
- understand the strengths and weaknesses of the method
- be aware of the major application domains and their appropriate design paradigms

Enable the student to evaluate software design:

- appropriateness of method
- completeness and correctness of design
- traceability to requirements and code
- design quality
- formal verification of design

Philosophy of the Course

The overall philosophy is that software is a *product of value* that is built in response to a need, that is to be maintained as long as the need persists, and that is to evolve as circumstances change.

Syllabus

The syllabus assumes 28 class meetings, including midterm and final examinations. Each meeting is planned to include approximately 55 to 60 minutes of lecture and 20 minutes of class discussion.

1. The Software Development Process and the Role of Design
2. Classification of Design Methods
3. Basic Principles of Design Methods
4. Statecharts: Description
5. Statecharts: Worked Example
6. MASCOT: Description
7. MASCOT: Worked Example
8. Object-Oriented Design: Description
9. Object-Oriented Design: Worked Example
10. VDM: Description
11. VDM: Worked Example
12. Comparison and Critique of Design Methods
13. Team Exercise: Presentation
14. Team Exercise: Requirements Review
15. Midterm Review
16. Midterm Examination
17. Application Domains and Design Paradigms
18. Information Systems Design: Problems
19. Information Systems Design: Paradigms
20. Real-Time Systems Design: Problems
21. Real-Time Systems Design: Paradigms
22. Team Exercise: Design Reviews
23. Team Exercise: Design Reviews (Continued)
24. Software Development Environments
25. User Interface Design: Problems
26. User Interface Design: Paradigms
27. Final Review
28. Final Examination

Summaries of Lectures

1. The Software Development Process and the Role of Design

Reading: none

The course is structured in two major parts: understanding design methods, and the application of those methods. The first part covers the software process and the role of design, and the classification, underlying principles, and verification techniques for several methods. The second part covers assessment and comparison of methods, their strengths weaknesses, their application domains, and paradigms for their use.

A software design is a description of a system that will meet the requirements. The design should be as complete as possible, and it is often represented formally using special notation. The process of design consists of taking a specification (a description of what is required) and converting it into a design (a description of what is to be built). A good design is complete (it will build everything required), economical (it will not build what is not required), constructive (it says how to build the product), uniform (it uses the same building techniques throughout), and testable (it can be shown to work).

The design activity is one part of a development process, and thus exhibits the characteristics of all activities: it occurs at a particular stage, it takes inputs from the previous stage, it performs a specific task, it generates outputs to the subsequent stage, it yields feedback to earlier stages, it incorporates feedback from later stages, and it is assessable both independently and within context.

A design can be assessed independently by asking questions such as these. Is it clear what is to be built? Is it clear how it is to be built? Are the components mutually consistent? Are the building techniques mutually compatible? Most of these questions look at the design representation—the object that embodies the design. A design can also be assessed in the context of the process by asking questions such as these. Does the design address every part of the requirement? Does the design meet all constraints on the solution? Are specific design *choices* clearly recorded? Can the design be effectively implemented? Is the design robust against small changes in requirements? Most of these questions look at the design documentation—the objects that accompany the design and explain it.

Traceability is the single most important concept in software development. From a component of a design (or other) document, one should be able to determine *why* that component is present, a *corresponding* component in another document, the *history* of the component across versions, *alternative* forms of the component across configurations, and the *reasoning* that created the component.

It is not possible mechanically to deduce a design from a requirement. Design is therefore necessarily a human activity. It consists of the intelligent application of expertise to the solution of problems and the realization of the solutions as software artifacts. This intelligence can be guided by design methods and techniques, but it cannot be replaced.

The development of large software systems, often called programming-in-the-large, is an activity very different from programming-in-the-small. In the latter case, programmers write programs, but in the former, teams build software. This required different ways of thinking, different ways of behaving, and different tools and techniques. Team development of software is characterized by the division of labor, replacement of individuals, more thorough specification of components and interfaces, more (and more formal) record keeping and communication, clearer separation of development stages, and much tighter control of feedback and consequent changes.

When a team performs design, control is required to ensure consistent notation, division of work without gaps or overlap, and consistency between components. The overall design must remain within the solution envelope, which places limits on complexity and on resource consumption.

2. Classification of Design Methods

Reading: Firth87

Classification of design methods is helpful as an aid to explanation of design methods, a help in understanding how methods differ, and a guide to selection of appropriate methods. Classification can use several kinds of criteria. Attributes of the design object are the simplest criteria to apply. They include the kind of design notation used (graphical, textual, mixed), degree of formalism (informal, such as natural language or fuzzy blobs), semi-formal (such as stylized phrases, shapes, and icons), or formal (logical propositions, directed graphs). Attributes of the structure of the representation are another type of criteria. They include whether the design is hierarchical (each level expands into more detail) or flat (there is one key level that is unique), and whether it provides a single view of the design or multiple views of the same design. Attributes of the design philosophy are perhaps the major guide to classification.

Much of the philosophy of a design method hinges on the view it takes of the system The basic view of the system taken by a design method, and hence captured by a design based on that method, can be functional, structural, or behavioral. With the functional view, the system is considered to be a collection of components, each performing a specific function, and each function directly answering a part of the requirement. The design describes each functional component and the manner of its interaction with the other components. With the structural view, the system is considered to be a collection of components, each of a specific type, each independently buildable and testable, and able to be integrated into a working whole. Ideally, each structural component is also a functional component. With the behavioral view, the system is considered to be an active object exhibiting specific behaviors, containing internal state, changing state in response to inputs, and generating effects as a result of state changes. This view is essentially dynamic rather than static. This is the basis of the classification scheme in Firth87. Observe that it can be applied to more than just the design stage.

Attributes of the design process are also useful in classifying methods.

A design method can be categorized also by the way in which it is used. The principle criterion is the manner in which the design object is supposed to be generated. This incorporates the design method's view of the design process and how it should be carried out. Some criteria imply alternatives—either this or that. Some criteria imply gradations—either more or less.

Design methods seem to have three views on how the design is to be generated. The top-down approach is to create a single, high-level description of the system, then take each component in turn, elaborate its interfaces, and refine its internal details. This process continues hierarchically from component to subcomponent. The bottom-up approach is to design in detail the lowest-level components, define the manner in which they can be composed, and then build higher-level components in terms of the lower-level ones. This continues until the highest component—the system—is reached. The left-right approach is to take a plausible scenario, define the information objects that flow through the system from initial input to final output, define the processing nodes through which the information will flow, and define the data stores that will hold information between scenarios.

In addition, some methods encourage prototyping and reuse. Prototyping methods provide tools and techniques for rapid creation of user interfaces; simulation of unimplemented components; scenario generation, recording and analysis; and animation of designs. The advantages are that the designer gets user feedback at an early stage, and the designer gets an assessment of how well the design fits within the specification constraints. Reuse methods provide libraries of standard components, ways to create new components and categorize them, ways to adapt or customize components, standard ways of interfacing components, and standards for data validation and error handling. This allows a designer to reduce implementation and testing costs.

There is some correlation between the design process and the view the method takes of the system. A functional view suggests top-down design; a structural view suggests bottom-up design; and a behavioral view suggests left-right design. There is also some correlation with application domain, a topic for later discussion.

3. Basic Principles of Design Methods

Reading: none

Design principles help us understand design methods, apply design methods, grasp the underlying nature of software design, and provide a framework with which to analyze the design process. There are two basic types of design principle: static concepts, which are things observable in design objects, and dynamic concepts, which are things observable in the design process. A design object exhibits structure, modularity, abstraction, information hiding, and hierarchy. The evolution of a design goes through three stages: derivation of a design object, examination of a design object, and generalization and subsequent reuse of a design. Each system component passes through all stages, but not all components are at the same stage at the same time.

Derivation of design is the process of creating a design object. It employs the principles of partitioning, deferral, refinement, composition, and elaboration.

Partitioning is the division of a problem into components, each with conceptual integrity, each simpler than the whole problem, each with a defined interface, and capable of being combined into a solution. A proper partitioning yields a modular design.

Deferral is conscious postponement of design issues: because they are global and cannot be assigned to one partition; because they are irrelevant and have no effect on this stage of design; or because a design choice requires more information.

Refinement is adding more detail within an existing framework, with no extension to prescribed functionality, and with no change in defined interface. This involves making specific design choices: analyzing alternatives, selecting the most reasonable, and documenting the result. Refinement usually creates a lower level of detail.

Composition is combining already-designed objects into an object at a higher level, with consistency of interfaces, common understanding of data representations, and cooperating functionality. Refinement and composition processes help build hierarchical structures.

Elaboration is adding extra functionality at a specific level (features previously deferred or features newly required) while maintaining the integrity of the whole (unchanged interface and original functionality undamaged).

Examination of design is the process of looking at a finished design object by verification, analysis, animation, execution, and quality assessment.

Designs and design objects of wide applicability are created by generalization, specialization, customization, enhancement. Such designs and design objects are candidates for subsequent reuse.

4. Statecharts: Description

Reading: Harel87

Statecharts are a notation for capturing design. They exhibit a behavioral view, and they are based on finite state machines. They use a graphical notation and display a limited hierarchy. They are incorporated in the Statemate system development environment supported by i-Logix Inc.

[The lecture includes a detailed description of statecharts.]

5. Statecharts: Worked Example

Reading: Bruns86

The problem for this example is the design of a simple elevator system. Inside the elevator there is a floor visit button for each floor, whose purpose is to cause the elevator to visit that floor. There are two elevator request buttons on each floor (up and down), whose purpose is to cause the elevator to visit that floor and then move in the appropriate direction. Requests are cancelled by appropriate visits. Of course, the doors must be opened and closed appropriately.

System inputs are floor visit requests (entered by pressing the button Visit(f) for floor f), elevator requests (entered by pressing Up(f) or Down(f) on floor f), and elevator location data (determined by an AtFloor(f) sensor near floor f). System actions are specified for elevators (ascend, descend, stop), doors (open, close), and visit buttons (cancel).

The system goal is to be able to cancel all buttons and go back to sleep. An implementation goal is to have a car reverse direction as infrequently as possible.

[The lecture develops the solution to this problem in detail.]

6. MASCOT: Description

Reading: Jackson84

MASCOT is an acronym for Modular Approach to System Construction Operation and Test. It was devised during 1971 to 1975, and is described in Official Definition of MASCOT (MASCOT 1) 1978; The Official Handbook of MASCOT (MASCOT 2) 1983; The Official Handbook of MASCOT (MASCOT 3) 1983. MASCOT has been a required UK MoD software design method since 1981.

The development of MASCOT has addressed these issues: the proper design philosophy for real time systems; use of a clear and consistent standard design notation; reuse of components of real-time systems; automatic generation of code from designs and templates; and systematic testing and profiling of real-time components. It exhibits a structural view of a system and is based on dataflow concepts. It uses a graphical notation and precise naming conventions. The design can be translated into code skeletons.

7. MASCOT: Worked Example

Reading: none

The problem for this example is the design of a message transfer system that is circuit based, with connections made and broken dynamically, and that allows messages to be transferred between connected nodes. The design also should include the data handling component: lines built up from input characters; lines broken down into output characters; and line buffers recycled around the system.

[The lecture develops this example in detail.]

8. Object-Oriented Design: Description

Reading: none

Object Oriented Design (OOD) attempts to partition the requirement into components, determine component relationships, classify the components, derive solution components, and implement and integrate solution components. It uses a single paradigm—the *object*. An object possesses some or all of state, behavior, and attributes. It captures a concept or aspect of the requirements. It may be an instance of a more general class, and it may perceive other objects in the system.

The approximate order of progress in the basic OOD method is inspect the requirements; identify the objects; identify the states, attributes and operations; identify the object interconnections; classify the objects; specify the object interfaces; implement the objects; and build the solution.

9. Object-Oriented Design: Worked Example

Reading: none

The example is the document concordance problem from Booch83. The solution method is to identify objects, identify operations, establish visibility, classify the objects, and then establish the interfaces. At this point, the iconic graph can be drawn.

Identifying the objects begins with the simple step of underlining the nouns in the requirement, resulting in: concordance (1 occurrence), document (4), entry (4), line (4), occurrence (3), page (4), and word (5). This suggests that the four major objects (and their subcomponents) are document, concordance (entry), word, and occurrence (line, page).

The document is seen as an external file, so operations include *open* and *close*; there is a need to read it one word at a time, so other operations needed are *at_eof* and *get_next* (note that the *get_next* operation must yield both a word and an occurrence, which is a page and line number). The concordance is also an external file, so operations include *open* and *close*; it is built by adding one entry at a time, requiring an operation *add_entry*; and because we may be required to print it out, a *print* operation is needed. We must create the concordance in alphabetic order, so we add an operation "<" on words; in order to ensure each word appears only once, we also need a "=" operation; to print the word, we need a *print* operation. We need to create a list of occurrences in document order, so we need a "<" operation on occurrences; we probably need to print the list, so we need a *print* operation.

Establishing the visibility of one object class by another is really done in parallel with the previous step. Document sees word and occurrence; concordance sees word and occurrence; word sees nothing; occurrence sees nothing. Finally, the requirement specifies a root component *make_concordance* that sees everything.

The design presented here diverges from the reference in that it adds ordering operations on words, uses occurrence as an abstract data type, and adds print operations for concordance components. The proper set of operations is not obvious. Objects cannot be elaborated in isolation—look especially at the operations on words.

Finding the objects in even this simple example is not automatic. Is "entry" (a word related to a page and line) a top level object? Is "occurrence" an object, or should we use page and line? Why isn't "list" an object?

The solution has some defects or omissions: error conditions and error handling; and identification of actual files to be opened. There are also some open issues: who understands the lexis of a word; that is, what constitutes a word? Also, who is responsible for formatting the output?

10. VDM: Description

Reading: Andrews87

The Vienna Development Method (VDM) was developed in the 1970s at the IBM Vienna Research Laboratories. It was influenced by 1960s work that produced the Vienna Definition Language (VDL). It has been applied in programming language definition, database design, and office automation systems.

The formal specification language used in VDM is Meta-IV. It includes elementary types (boolean, numeric, quotations, tokens), composite types (sets, lists, tuples, maps, records), and combinators (clauses, statements, blocks). Naming conventions are also followed. Fundamentals of Meta-IV specifications include semantic domains, invariants, and functions.

The VDM method is based on stepwise refinement (decomposition and reification) and proof obligations (at each step of refinement, define a *retrieve* function, prove its adequacy, and prove commutativity of operations with respect to it).

[The lecture includes examples of Meta-IV syntax, a Meta-IV specification of a stack, and a reification example.]

11. VDM: Worked Example

Reading: none

An example of VDM is the design of a company's security system. The system should record the names of employees who enter and leave the building, doors have badge readers connected to the computers, and the doors can be opened by the computer. Basic operations include *init()* {building is empty}, *enter(nm: Name)* {name read from card}, *exit(nm: Name)* {name read from card}, and *is-present(nm: Name) r: Bool* {check on name}. The basic model defines *Work-force = set of Name* {those who are in the building}, and *Name = (* to be defined *).*

Subsequently, new requirements are defined: to select a list of volunteers, and to count how many employees are present. Data reification results in a new model *Work-force1 = seq of Name* and revised basic operations. [The lecture develops the example in detail.]

A summary of the method is (1) choose a more concrete data representation *Rep*; (2) discover the retrieve function: *retr: Rep → Abs;* (3) prove adequacy; (4) rewrite the operations; (5) prove that the invariant is preserved; and (6) prove acceptability and commutativity. Some observations about the method: concrete representations introduce too many values, but invariants kill off the excess values; invariants also represent common knowledge about the state of the system; the method emphasizes the ability to state requirements at each level of the design in rigorous, mathematical terms.

12. Comparison and Critique of Design Methods

Reading: Bergland81, Yau86

Common features of most design methods include some special design notation, procedures for looking at requirements, procedures for creating designs, ways to analyze designs, and some possibility of tool support.

Design notations seem to evolve in the same way: a basic notation for aspects considered primary, followed by elaboration to capture more information, and then annotations to overcome defects in the basic notation. A tentative conclusion is that no single notation is adequate.

Most design methods start from the requirements as given, they assume a certain style of specification, expect design components to be extracted from it, and verify design correctness against it. Two kinds of approach to analysis are component oriented and scenario oriented. Some design methods expect to refine the requirement.

Design creation is usually incremental—composed of distinct stages. Design methods require differing degrees of component isolation and backtracking, and they differ in how strictly they try to control this process.

Design analysis serves different purposes: to verify proper design; to deduce properties about the system being designed; to determine implementation strategies; and to determine implementation costs. Design methods provide different amounts of help to the implementor.

Some design methods require tool support: the notation is impossible to maintain by hand; numerous consistency checks must be repeated; or rapid prototyping requires automation. Some design methods resist tool support: the notation is largely irrelevant to the design process; design properties can be assessed only informally; or the behavior of a prototype cannot be deduced from the design.

There are several key differences in philosophy among design methods, including terminology (domain based or method based); order of work (linear or spiral); and reuse (by afterthought or by forethought). There are also differences in process that arise from three causes: differences in philosophy, differences in view of the life cycle, and differences in emphasis on design goals.

Philosophy can affect the process in these ways: special terminology forces early abstraction; linear work order implies hierarchical design; and emphasis on reuse encourages depth-first elaboration. The view of the life cycle can affect the process in these ways: the waterfall model implies that requirements are taken as given and complete, a complete design must precede implementation, and design verification rarely addresses feasibility. The spiral model implies that requirements are subject to refinement, a partial design should be prototyped or simulated, and design feasibility can be tested on a prototype. Most design methods still use the waterfall model. Design goals can affect the process because design methods presume diverse goals: does the design criticize the requirement; should the design imply implementation strategies; will the design be reviewed by the user of the system? These questions imply differences in how the design is done and what kind of design objects are created.

Different views of design have different advantages. With a behavioral view, it is easy to create scenarios for user criticism; capturing time and event-dependent properties is easier; and exhaustive testing of required invariants is allowed. With a structural view, it is possible to identify implementation components, distinguish active from passive components, distinguish between persistent and dynamic data. A structural view also encourages reuse and allows subcomponents to be prototyped and tested. A functional view is easiest to verify against the requirement, uses mostly domain-related terminology, and makes later enhancement simpler. A formal view builds on a very secure foundation, resists introduction of arbitrary assumptions, and proceeds by small and verifiable steps.

The strengths of design methods are that they generally help by introducing an agreed notation and terminology, provide rules for creating designs, provide ways to analyze designs, and act as a basis for tool support. The weaknesses of many design methods are that they promise too much, do not capture reasons for design decisions, try to force all problems into one framework, and fail to deal with holistic properties of the system (such as response time or storage consumption).

General conclusions are that there is no way to automate fully the process of design. Design methods are fundamentally a way for the designer to organize work, an aid to accurate and appropriate thinking, and a way to capture design in a communicable form. Design methods should treat differently the creative and the mechanical parts of the design activity.

13. Team Exercise: Presentation

Reading: Floyd86

The team exercise is to be done during the second half of the course, and it forms part of the course assessment. It presents a problem in a major application domain for which case studies already exist. It provides some practice in software design and builds on previous study of design methods. Students work in teams of three or four, and the final design is assessed as a whole and equal credit given to each team member.

The team exercise is based on Floyd86. The case study requirement will be used as the basis for a software design.

The exercise begins with a requirements presentation, and proceeds through a requirements review, design consultation, formal design review, design revision, and submission of finished design.

Goals of the exercise are that students will perform a simple software design by selecting a design method, applying a design method, and documenting the process and the result. They will practice team working, division of work, and integration of work. They will also perform a software design review.

14. Team Exercise: Requirements Review

[This class is a requirements review for the team exercise. Its objectives are to clarify the requirements, agree on terminology, and confirm team composition. It ensures that all teams start with a common understanding of the requirement. Discussion by the teams is emphasized, with the instructor stepping in as needed to initiate discussion, and to raise and illustrate important issues. In the absence of a real customer, the teams will decide on “reasonable” amplifications and clarifications.]

15. Mid-term Review

Reading: Balzer85, Brooks86, Gomaa88, Parnas86

16. Mid-term Examination

17. Application Domains and Design Paradigms

Reading: Hesse84

An application domain is a separate discipline with requirements that recur, involving a common set of objects or issues, and with its own technical expertise. For example, within engineering, there is a separate domain of bridge building. Domain specific problems are those that recur in the majority of systems requirements, seem to be implied by the domain itself, and address fundamental issues in building solutions. The problems in the domain can affect the design process or design method used.

The term *paradigm* means a standard solution or solution method applied to a domain specific problem that can be independently taught and applied, and that captures a major engineering insight.

The existence of paradigms influences design. First efforts at solving a new problem are unorganized. Gradually, solutions begin to exhibit common factors, and eventually, some of these factors are isolated. A small number of standard solutions becomes part of the engineering expertise. The common problem factor and its standard solution then become a paradigm.

Once one knows a set of paradigms, one's attitude to new problems changes. One tries to relate the problem to one of the standard problems, in order to apply a paradigm. This is very helpful if the problem is indeed another example of the familiar type. But it can be a serious handicap if in fact the problem has genuinely new aspects.

Paradigms can be explored and developed in isolation. They represent sets of solutions to common problems. They can lead to reusable components applicable to new problems in the same domain. The properties of the solutions can be assessed and used to help select the right solution in each case. Access control mechanisms have reached this stage.

The existence of paradigms influences the way requirements are analyzed and hence the way the solutions are designed. The analysis tends to search for familiar problem components, and the design tends to structure the solution around the key components for which paradigms are available. For example, in the construction of operating systems, one fundamental paradigm is that the function of the operating system is to allocate system resources to user processes. Fundamental principles include: resources represented by objects are allocated to processes, competition is arbitrated by a single scheduler, and processes wait until resources are available. The design of the operating system tends to focus on: the resources and their representation, fair and efficient allocation mechanisms, robust control of resources, and attributes of users or processes that affect their rights to resources.

18. Information Systems Design: Problems

Reading: none

Information systems manage data. Basic tasks include collection, validation, organization, processing, and retrieval. Fundamental concepts include data, representations, information, entities, attributes, and currency. Important problems are data format, data validation, entities and attributes, data consistency, data retrieval, and data presentation.

19. Information Systems Design: Paradigms

Reading: none

Each data type should be defined once, including name, explanation, language type, representation, display form, and constraints. This definition is written in a data definition language.

There are three main issues for entity identification: by what attribute are entities identified; how are attribute values assigned to new entities; and how is the attribute of the current entity captured.

The attribute that uniquely identifies an entity to the system is called its key. Attribute assignment, or determining to which entity an attribute belongs, is a fundamental question. In the examples of the relationship of car to owner's address, or of library book to borrower's office, the attribute does not belong. The owner may move and take the car, or the borrower may move and take the book.

There are two problems with maintaining current data: some data change spontaneously (an example is a person's age), and some data are derived from other data (an example is total sales). How can one ensure that all data remain current? The update consistency problem arises when two or more data are related (such as in the case of "spouse of," "employee of," or "employer of" relationships). An update must leave both data in a consistent state. In particular, system failure must not cause inconsistency.

Data retrieval usually follows the relational model, with operations select (choose a subset of the set of entities); project (choose a subset of the set of attributes); join (combine attributes of one entity with those of a related entity); prune (remove duplicate information) count (determine cardinality of a subset); sum (total values of an attribute across a subset); and sort (arrange the members of a subset in order). Every enquiry is phrased in terms of these operations

Some effects on system design are a need for clear definition of data types, reuse of standard types, careful identification of entities and attributes, and an awareness of consistency and currency issues. Other important ideas are the use of scenarios based on a transaction model, that robustness is an integral part of system operation, that retrieval operations are analyzed systematically, and that the system must be tuned for most frequent operations.

20. Real-Time Systems Design: Problems

Reading: Gomaa86, Kelly87

Real-time systems exhibit requirements involving time, response to external events, parallelism, and time-dependent behavior. Examples include: measure vehicle position every 100 ms; reset all clocks on the hour; respond to a signal within 10 ms; process signals at a peak rate of 100/sec. The magnitude of the time interval varies; what matters is the importance of keeping to it.

Real-time systems typically interact closely with their environment; examples include: whenever the telephone rings, answer it; whenever a button is pressed, light it; whenever a key is pressed, read a character; whenever the vehicle orientation changes, update the heading data. The occurrence of these events is unpredictable.

Fundamental concepts for real-time systems include time, events, processes, and synchronization. Major problem are representation of time, representation of events, implementation of parallelism, distribution of processing across processors, robust guarantees for timely response, and testing of time-dependent systems.

21. Real-Time Systems Design: Paradigms

Reading: none

Some issues and paradigms include time as the basis of design, events as the basis of design, parallelism and synchronization, design for functional correctness, design for trustworthy performance.

22. Team Exercise: Design Reviews

[This class period and the next are devoted design reviews for the student team exercise.]

23. Team Exercise: Design Reviews (Continued)

24. Software Development Environments

Reading: Dart87

A software development environment (SDE) is an integrated set of methods, processes, and tools for the orderly construction of software; usually there is substantial automated support. An SDE serves several purposes: consistency, economy of effort, speed of production, transition of product, productivity of staff,and management of process.

The following features are usually found in an SDE: an object base, a command interface, a tool set, training aids, and a component library.

The major advantages are clear: more efficient use of human effort; more robust, better organized and better tested products; consistency within products and between them; easier transition of products to an equipped site. There are some corresponding disadvantages: expensive hardware and software support; SDE usually tied to a single design method; hard to incorporate familiar or third-party tools; reliance on one vendor for most tools; and the fact that the transition site must buy in to SDE to maintain the product.

A software development environment can have several effects on the design process. It usually mandates a specific design method. It allows larger projects to be organized and managed. It allows more exploratory design and prototyping, and supports a growing base of reusable components. It smooths the transition between development and maintenance.

25. User Interface Design: Problems

Reading: Myers89

[Guest lecturer: Brad Myers, Carnegie Mellon University]

A user interface is the end-user-visible parts of software: all inputs from the user to the computer, and all outputs from the computer to the user.

What does it mean to be a *good* user interface? Some proposed definitions are : “I like it,” “I always do it that way,” “That is the way the xxx system does it,” “It is easy to implement.” Much better are definitions such as: it can be learned in less than 20 minutes; the error rate will be lower than 1 per 40 operations; tasks will be performed in 30% of the time it took before the system was used; users will have a high satisfaction with the system as measured by a survey.

Some attributes of good user interfaces are that they are invisible, have minimal training requirements, people begin doing real work quickly, high transfer of training, predictability, easy to recover from errors, people perform real tasks well, experts operate efficiently, it is flexible, people like it.

Why are user interfaces hard to design? There are many different aspects to the design that need to be taken into account: graphics (artistic design), human factors principles, dialog control issues, implementation constraints, efficiency constraints, standards, analysis of existing systems. It is very hard to "know the user," and it may take years to learn the application area. Problems with users include that they don't care about elegance, sophistication, or complexity of the design, only what it can and can't do for them; they come in various levels of sophistication and experience; they will do the unexpected; they will refuse at all costs to read the manuals; they will fail to observe even prominent instructions; and they won't ask for help when they need it. Problems with the software designers include that they assume too much about users; they assume users are just like them; they are sure that people can learn to use anything; they get "attached" to hardware and software; they may never use the system they design; and they are suspicious of "soft" sciences like psychology and human factors.

Global problems for user interface designers include that standards are adopted (and rejected) too quickly, and for political or economic reasons; economic incentives often promote the status quo; and there is a tendency to use old techniques on new technology. It is not sufficient just to use icons and windows. Graphics can sometimes be worse; some experiments show that there may be no significant improvement with spatial (location) cues versus symbolic (name) cues on information retrieval tasks; other experiments show that naive users performed more poorly with iconic systems than with command or menu systems on filing tasks and that users performed more slowly (but more accurately) in a windowed environment compared to a non-windowed environment on editing tasks. It is not sufficient just to use a standard look and feel; the standard does not usually address all issues (such as insides of windows). Reasonable people can differ, so there is a need to test real users. This usually implies a need for iterative design. Legal problems with copying the "look and feel" of a user interface also exist.

User interfaces are hard to implement because of the need for iterative design; the need for prototyping; the difficulty of getting the screen to look pretty; and asynchronous inputs. Usually there is a need for multiprocessing to deal with user typing, window refresh, and multiple input devices. There is a need for efficiency in order to address requirements such as output 60 times a second, keeping up with mouse tracking, or real-time programming. There is a need to handle prompts, feedback, errors, help, and aborting. User interfaces are often hard to modularize; the distinction between what is user interface and what is application is not clear. There is little language support; primitives in the programming language make bad user interfaces.

There are several user interface styles, meaning the method for getting information from the user, and the choice of style will have a big impact on the software design. Some styles are question and answer, single character commands and/or function keys, command language, menus, forms/dialog boxes/property sheets, editing paradigm, direct manipulation, WYSIWYG, gestures, and natural language. Usually, interfaces provide more than one style, such as a command language for experts with menus for novices, or menus plus single characters (Macintosh). The appropriate style depends on the type of user and the task.

Major issues in user interface style include who has control, ease of use for novices, speed of use (efficiency) once the users become proficient, generality/flexibility/power (how much of the user interface will this technique cover?), ability to show defaults and current values, and skill requirements required (such as touch typing).

[The lecture includes discussion of an in-class experiment in human factors that can affect user interface design.]

26. User Interface Design: Paradigms

Reading: none

[Guest lecturer: Brad Myers, Carnegie Mellon University]

Components of user interface software normally include a window manager, a graphics package, a user interface toolkit, and a user interface management system.

A window manager manages and controls multiple contexts by separating them into different physical parts of the screen. It can be part of operating systems, a separate program, or part of a program. It provides output graphics operations to draw clipped to a window, and channels input from a mouse and keyboard to the appropriate window.

A graphics package provides the interface for doing graphics and getting input. It is often standardized; it may be device independent; and it can exist in the window manager, on top of the window manager, or instead of the window manager.

A user interface toolkit is a library of interaction techniques that can be called by application programs; an interaction technique is a way of using a physical input device to input a certain type of value. Toolkits contain procedures to do menus, scroll bars, buttons, window decorations, dialog boxes, etc. Normally, no higher-level or sequencing control is provided, so the toolkit can be hard to use.

A user interface management system (UIMS) helps the programmer create and manage many aspects of the user interface. It may (or should) handle all input, validate user inputs, handle user errors, handle aborts and undo, provide appropriate feedback for input, provide help and prompts, help with screen layout and graphic design, handle the updating of the screen when data changes, notify an application when the user changes graphics, deal with scrolling and refresh, handle the sequencing of operations, insulate the application from the window manager, allow end-user customization of user interface, and automatically evaluate a user interface and suggest improvements.

Window manager output models include a simple terminal model, a collection of functions to be called, and a programming language. A major issue is device independence.

Window manager input models include polling (which is inefficient, has no type-ahead or “mouse-ahead”), but all modern systems use same technique: events. An event manager maintains a queue per window for all events (keystroke, mouse action, etc.). This is currently the best technique known.

Toolkits (such as the Macintosh Toolbox, NeXTStep, and Xtk) may be object-oriented or just a collection of procedures.

Components of user interface management systems include a toolkit; a design-time component for the designer to specify the interface, such as some kind of compiler or editor; and a run-time component to operate the interface at run-time (usually libraries of routines that the code links to).

Interface aspects covered by a user interface management systems include the design of graphical components: choosing the components that are in the interface, graphical layout of existing components, sequencing of actions, "insides" of application windows, and evaluation of the interface.

User interface management system styles include menu networks (simple trees or graphs of menus), state transition networks (a picture of the sequence of actions), grammars (only good for command language syntax), event languages (input tokens are events that are sent to handlers, which then cause side effects and send new events), declarative languages (that specify what should happen or what the contents are, not how to do it), object-oriented languages (the system provides higher level classes that the designer specializes), and direct graphical specification (define an interface by placing objects on the screen showing what the end user will see).

A major design issue is communication: how the application and the user interface exchange information. Solutions include the application calls the user interface procedures; the user interface calls application procedures; mixed control where both allowed; and parallel control. A related issue is how often the application and user interface exchange information.

A process for building a user interface is: learn the application; learn the users; learn the hardware and environment constraints; evaluate user interfaces of similar products and other products for the same environment; determine the support tools available (toolkits, user interface management systems); plan for more than one user interface in the schedule and budget; involve graphic artists, user interface professionals, and technical writers (for documentation); plan to incorporate *undo*, *cancel*, and *help* from the beginning; try to make the user interface as "direct" as possible; follow user interface guidelines in design; separate the user interface from the application; design the application assuming the user interface will change; use object-oriented architecture; and prototype and test with actual end users.

27. Final Review

28. Final Examination

Bibliography

Andrews87	Andrews, Derek. "Data Reification and Program Decomposition." *VDM '87: VDM—A Formal Method at Work. VDM-Europe Symposium 1987*, Bjørner, D., Jones, C. B., Mac an Airchinnigh, M., and Neuhold, E. J., eds. Berlin: Springer-Verlag, 1987, 389-422.
Balzer85	Balzer, Robert. "A 15 Year Perspective on Automatic Programming." *IEEE Trans. Software Engineering SE-11*, 11 (Nov. 1985), 1257-1268.
Bergland81	Bergland, G. D. "A Guided Tour of Program Design Methodologies." *IEEE Computer 14*, 10 (Oct. 1981), 13-37.
Booch83	Booch, Grady. *Software Engineering with Ada*. Menlo Park, Calif.: Benjamin/Cummings, 1983.
Brooks87	Brooks, Frederick P., Jr. "No Silver Bullet: Essence and Accidents of Software Engineering." *Computer 20*, 4 (Apr. 1987), 10-19. Originally published in *Information Processing 86*, H. J. Kugler, ed. Elsevier.
Bruns86	Bruns, G. L., S. L. Gerhart, I. Foreman, M. Graf. *Design Technology Assessment: The Statecharts Approach*. Tech. Rep. STP-107-86, MCC, Austin, Texas, 1986.
Dart87	Dart, Susan A., Ellison, Robert J., Feiler, Peter H., and Habermann, A. Nico. "Software Development Environments." *Computer 20*, 11 (Nov. 1987), 18-28.
Fairley85	Fairley, Richard. *Software Engineering Concepts*. New York: McGraw-Hill, 1985.
Firth87	Firth, Robert, Wood, Bill, Pethia, Rich, Roberts, Lauren, Vicky Mosley, and Dolce, Tom. *A Classification Scheme for Software Development Methods*. Tech. Rep. CMU/SEI-87-TR-41, ADA200606, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 1987.
Floyd86	Floyd, Christiane. "A Comparative Evaluation of System Development Methods." *Information Systems Design Methodologies*, Olle, T. W., Sol, H. G., and Verrijn-Stuart, A. A., eds. Elsevier, 1986, 19-54.
Gomaa86	Gomaa, Hassan. "Software Development of Real-Time Systems." *Comm. ACM 29*, 7 (July 1986), 657-668.
Gomaa88	Gomaa, Hassan. *ADARTS — An Ada-based System Design Approach for Real-Time Systems*. Tech. Rep. SPC-TR-88-021, Software Productivity Consortium, Reston, Va., Aug. 1988.
Harel87	Harel, David. "Statecharts: A Visual Formalism for Complex Systems." *Science of Computer Programming 8* (1987), 231-274.

Hesse84	Hesse, Wolfgang. “A Systematics of Software Engineering: Structure, Terminology and Classification of Techniques.” *Program Transformation and Programming Environments*, Pepper, Peter, ed. Berlin: Springer-Verlag, 1984, 97-125.
Jackson84	Jackson, Ken. “MASCOT.” *IEEE Colloquium on MASCOT.* New York: IEEE, Dec. 1984, 1/1-1/14.
Kelly87	Kelly, John C. “A Comparison of Four Design Methods for Real-Time Systems.” *Proc. 9th Intl. Conf. Software Engineering.* Washington, D.C.: IEEE Computer Society Press, 1987, 238-252.
Linger79	Linger, Richard C., Mills, Harlan D., and Witt, Bernard I. *Structured Programming: Theory and Practice.* Reading, Mass.: Addison-Wesley, 1979.
Myers89	Myers, Brad A. “User-Interface Tools: Introduction and Survey.” *IEEE Software 6*, 1 (Jan. 1989), 15-23.
Parnas86	Parnas, David L. and Clements, Paul C. “A Rational Design Process: How and Why to Fake It.” *IEEE Trans. Software Engineering SE-12*, 2 (Feb. 1986), 251-257.
Pressman87	Pressman, Roger S. *Software Engineering: A Practitioner's Approach*, 2nd Ed. New York: McGraw-Hill, 1987.
Ward86	Ward, Paul and Mellor, Stephen. *Structured Development for Real-Time Systems, Vol. 3: Implementation Modeling Techniques.* New York: Yourdon Press, 1986.
Warnier76	Warnier, J. D. *Logical Construction of Programs.* New York: Van Nostrand Reinhold, 1976.
Yau86	Yau, Stephen S. and Tsai, Jeffrey J.-P. “A Survey of Software Design Techniques.” *IEEE Trans. Software Engineering SE-12*, 6 (June 1986), 713-721.
Yourdon79	Yourdon, Ed and Constantine, Larry L. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* Englewood Cliffs, N. J.: Prentice-Hall, 1979.

3.4. Software Creation and Maintenance

Students' Prerequisites

Students are expected to have a computer science background, reasonable knowledge of programming, some experience of team software development, and some exposure to the software development process. Industrial experience is an advantage.

Objectives

Help the student make the transition from programming for the short term to programming for the long term. The student will:

- appreciate the software development process
- understand the role of creation and maintenance in that process
- understand how to analyze and implement a software design
- appreciate the need for software maintenance and evolution
- understand when and how to perform software maintenance
- learn how to plan for extended software life

Enable the student to create software from design. The student will learn to:

- analyze a design
- develop a "building plan" for the artifact described in the design
- develop a test plan and quality assurance plan for the artifact and its components
- select and employ the appropriate software creation processes
- build, integrate, and test the artifact
- deploy the created artifact

Enable the student to manage software maintenance. The student will:

- understand the basic purpose of maintenance
- appreciate the need for good maintenance processes
- understand the reasons for software change
- learn to employ effective mechanisms to control software change
- learn to apply proactive software improvement principles

Philosophy of the Course

The overall philosophy is that software is a *product of value* that is built in response to a need, that will be maintained as long as the need persists, and that will evolve as circumstances change. The course stresses that software development must follow a *process* from requirements to maintenance with proper quality assurance and traceability throughout. It further takes the view that *maintenance* is an integral part of the life cycle that must be considered during earlier stages, and that it operates not on the code but on *all* relevant documents. It includes *cost* considerations as a necessary driver of evolution.

The key concepts of the philosophy, relative to software creation, are:

- a software artifact must be built against a design
- this process must follow a proper building plan
- testing and assessment is necessary at each stage
- the entire process must be documented

Software creation ends with the act of *deployment*; what follows is maintenance. However, the *entire* design and creation history must be transferred to the maintenance organization. The purpose of maintenance is *to preserve the value of the software over time*. It is far more than mere error correction. It is driven largely by estimates of costs and benefits. The course discusses three kinds of maintenance: corrective, adaptive, and perfective. It also distinguishes between reactive and proactive maintenance.

Syllabus

The syllabus assumes 31 class meetings, including midterm and final examinations. Each meeting is planned to include approximately 55 to 60 minutes of lecture and 20 minutes of class discussion.

1. Introduction
2. Software Creation Overview
3. Analysis of Design Objects
4. Building Software Artifacts
5. Coding Styles and Standards
6. Software Reuse: Principles
7. Software Reuse: Practice
8. Automatic Software Generation
9. Software Prototyping Systems
10. Software Testing: Principles
11. Software Testing: Practice
12. Meeting Performance Constraints
13. Implementation Metrics and Costing
14. Midterm Review
15. Midterm Examination
16. Software Maintenance Overview
17. Causes of Software Evolution
18. Management of Software Evolution
19. Maintenance Tools and Environments
20. Dealing with Errors in Software
21. Requirements Evolution
22. Technology Evolution: Principles
23. Technology Evolution: Practice
24. Building Long-Lived Software
25. Software Quality Assessment
26. Reverse Engineering
27. Software Performance Improvement
28. Software Perfectability
29. Final Review (Part 1)
30. Final review (Part 2)
31. Final Examination

Summaries of Lectures

1. Introduction

Reading: none

The objectives of the course are to help the student make the transition from programming for the short term to programming for the long term; to understand the role of software creation and maintenance; to appreciate the need for evolution of software; to understand the need for cost-effective processes; and to learn how to plan for extended software life.

In particular, the course will enable the student to create software from design, which involves analyzing design, constructing software artifacts, assessing alternatives to coding from scratch, and testing and assessing software. It will also enable the student to manage software maintenance, including understanding the need for good maintenance processes, the reasons for software change, effective mechanisms for software change, and proactive software improvement.

The first part of the course addresses software creation, from design to deployment. This includes writing new software, software reuse, automatic software generation, software prototyping, testing performance assurance. and complexity and cost estimation. The second part of the course addresses software maintenance process, dealing with errors, requirements evolution, technology evolution, and preserving software quality.

2. Software Creation Overview

Reading: none

During this part of the course, we shall explore the software creation process, why that process seems so simple and yet is so hard, what are some effective ways to create good software, how to manage software creation, and what tools can help us. There will be some small exercises along the way.

The ideal process follows several steps: analysis of design, identification of structural components, realization of abstractions, construction or acquisition of components, testing of components, integration of components, testing of the system, and deployment. The process actually followed is rarely ideal—there is often a mismatch between function and structure, imperfect abstractions, pollution of design by target machine details, components with excessive difficulty or risk, or problems with ensuring "holistic" properties.

The goal in managing software creation is to be both effective and cost-effective, where *effective* means build a quality artifact as specified, and *cost-effective* means do so in a manner economical of people, time, and worry. A good software creation process will identify risks and difficulties early, address them promptly, and resolve them satisfactorily. It will use several technical and management tools, including prototyping and risk tracking.

Unfortunately, it is very tempting to defer problems. This gives an illusion of rapid progress, maintains cash flow, keeps morale high for a while, and keeps the customer happy. The illusion is abetted by the fact that people tend to underestimate the difficulty of problems not in their field.

A good process requires a well-structured environment: effective support for work, comprehensive capture of information, tools to help with technical work, tools to help with clerical work, and painless enforcement of standards. In sum, it requires a typical *software development environment.*

There are several ways of creating software. Design analysis identifies structural components, and then each component must be realized in some way. The conventional way is writing it from scratch. This is also the most expensive and risky way. In an effective software creation environment, writing software from scratch should be regarded as the last resort, not the first. There are many ways to avoid writing code: reuse an existing component, revise a similar component, generalize a similar component, generate the code automatically, or select or parameterize a component automatically.

3. Analysis of Design Objects

Reading: Prell84

Design analysis is a prelude to implementation. It guides the implementation, test, and build process. It involves specific decisions about *how* to build. The design method should produce a buildable design. It will not necessarily have identified structural components. However, it will have partitioned the requirements, elaborated on the objects and attributes involved, identified major interfaces, and specified resource constraints.

From inspection of the design, one should identify possible structural components, possible implementation layers, boundary conditions and constraints, problem areas, and test and integration strategies. Structural components include encapsulations of application-level and low-level objects; resource allocators; input receivers, transducers, routers, and output generators; algorithmic subroutines; and data stores and managers. For example, as part of an information system, one finds records and record types, queries, and consistency checks. As part of a real-time system, one finds hardware devices and drivers, message queues, and message types. For data processing applications, typical structural components are input reader, input verifier, command decoder, message router, query processor, report generator, output formatter, and output printer. Typical subroutines include mathematical functions, binary/ASCII conversions, sort routines, dictionary managers, statistical routines, curve fitting, and text justification. Typical data stores include conversion tables, data dictionary, dialogue menus, transaction log, and task schedule and time line.

One way to build a system is bottom-up: each layer is a level of abstraction; each layer uses the features of the lower layer; the bottom layer is the target machine; and the top layer is an abstract “application machine.” For example, a data base may be structured in these layers: (1) hardware mass storage device, (2) physical disc block IO, (3) disc storage allocator and reclaimer, (3a) disc verifier/diagnostic/repair program, (4) data base “realm” allocator, (5a) data dictionary manager, (5b) entity set manager. A graphical animator may be structured in these layers: (1) bitmapped video terminal, (2) window manager, (3) icon definitions; (3a) icon definition language/editor, (4) icon overlay manager, (5) icon motion generator/smoother, and (6) scenario player.

Boundary conditions represent the limits on the application objects, such as maximum size, maximum number, limit of value range, attribute existence, attribute uniqueness, relationship existence, temporal limits, and temporal ordering. Size constraints imply the amount of hardware and the choice of allocation and management algorithm. Existence and relationship constraints imply the structure of input verifiers and details of data store managers. Temporal conditions imply the use of active processes (daemons and alarms). Resource constraints represent the limits on the implementation: machine power, memory size, mass storage size, device latency time, and transaction response time. For example, the constraint of too small a memory may imply the need for code paging or overlays, data store paging, data queue paging, algorithms to have high locality, or queues to be accessed linearly. A time constraint such as device interrupt urgency may imply a high priority handler, special fast context switching, or minimal processing at the interrupt level. A time constraint such as transaction response limits may imply minimal data movement, pre-paging of code or data (very hard!), filtering of "easy" events, or background processing of other tasks.

The design analysis can also identify problems such as resource limits, eccentric and hard-to-abstract machine features, algorithms with variable performance, components that are stress points, and components that are hard to test. As an example, consider an "eccentric" device. Most output managers adopt the model "send message (destination, message text)." You have a hardware terminal handler that interrupts with "(line number of terminal now ready for next character)." It is not easy to layer the familiar model on top of it. Another example is an unpredictable algorithm. Many sort algorithms are have O(N log N) complexity. Some of these exhibit very bad worst-case performance, such as $O(N^2)$ on reverse-ordered input, and very good usual-case performance, such as O(N) on nearly-sorted input. Are we building for predictability or throughput? A *stress point* is a component, failure of which will destroy the system; examples are the garbage collector in transaction-processing system, the clock handler in real-time system, and the file server in distributed system. These components must be built very carefully. Components may be hard to test because of many different inputs, many different possible outputs from one input, output very dependent on past history, output time-dependent or nondeterministic, or distribution of inputs largely unknown.

The design analysis helps define the test strategy for each component: test from specification (black box), test from structure (white box), examine input and output, examine internal state, or measure performance. For example, a lexical scanner reads characters and emits tokens; it uses negligible internal memory, it has exact specification of input and output, its input characteristics are highly predictable, and the time taken is very predictable. A good test strategy is black box, from specification, covering the full input domain—test with every possible pair of tokens. As another example, consider a routine to compute the square root of a floating-point number: it has a precise specification and a domain with sharp boundary conditions, full test is impossible, and an accuracy constraint must be met. A good test strategy is white box, stressing the boundaries of domain and all flow paths through the code. It is also desirable to test sparsely over the entire range of input, paying attention to values close to a power of the radix.

These are simple examples from a wide field. The key point is that the build strategy of a component must include also a test strategy, which depends on the component, its role in the whole system, and the constraints under which it functions.

The result of the analysis is a building plan that describes what the components are, in what (partial) order they will be built, how each component will be tested, problem areas to investigate immediately, and the outline resource budget for each component. At this stage, one can investigate prototyping and reuse.

In summary, software creation begins with the software design. This design is analyzed to determine a build strategy. The result is a building plan traceable back to the design. Possible problem components have been identified. Resource requirements and constraints have been clarified.

4. Building Software Artifacts

Reading: none

Building software requires a building plan and a building team. Major concepts are the structure of the team, team management, code production and adaptation, tools and scaffolding, and configuration building.

The building plan contains the results of the design analysis (what the components are, in what (partial) order they will be built, how each component will be tested, and the outline resource budget for each component), and the instructions to the building team. The team must be organized to support several functions: product and resource management, quality assurance, code production, and configuration building. It must be managed efficiently and economically, and it must function within an appropriate framework.

The team manager is responsible for the overall work of the team and performs these duties: assigns resources, instructs in standards and procedures, monitors progress and budget, resolves implementation problems, handles emergencies, and keeps risks under control.

Quality assurance support is responsible for quality of product; it assesses compliance with standards and procedures, checks that all documentation and history are present, ensures that the established build and test plan is followed, performs spot checks as necessary, and certifies each component to librarian.

Work groups are responsible for building and testing components: code production, code adaptation, test set generation, internal checking of code quality, and internal monitoring of progress and resources.

The librarian is responsible for building final configuration, maintains appropriate information structures, accepts components when built and certified, ensures integration only of consistent versions, controls access to released documents, ensures proper revision procedures are followed, and maintains versions and configurations.

The work is assigned to the work groups by considering the order defined in the building plan, the skills of group members, the availability of resources, and the tightness of the schedule. No initial assignment survives reality. Progress tracking is essential. The order of work should be adaptable to adjust for unforeseen difficulties, accommodate gaps in resource availability, allow reasonable slack after each major activity, provide time for review and reconsideration, and have clear and unambiguous dependencies. It should allow the work groups to manage their own time as much as possible.

There are three keys to executing a successful build: regular progress tracking, rapid and effective response to emergencies, and continuous monitoring and re-evaluation of risks. The ideal is never to be surprised. The next best is to be equipped to deal with surprises.

Resources are the cost inputs to the process: people, equipment, and time. Resources available must be checked against estimates. Resources consumed must be checked against estimates. Major discrepancies must be resolved by reallocation. Only objective monitoring will convince the manager of the need for reallocation. Progress tracking measures the output value of the process: functioning components, correct documentation and history, and complete configurations. Progress made must be checked against estimates. Major discrepancies must be resolved by rescheduling. Progress must be measured objectively and, preferably, independently of the work groups.

Alerts and *exceptions* are unanticipated events that affect the process, such as people problems, equipment failures, or customer interference. The response must try to keep the process on track, and it might include reallocation of resources, adjustment of schedules and work order, or reconsideration of priorities. The manager must recognize *failure* of the process.

Risks are *anticipated* problems with the process, such as novel or difficult tasks, or major uncertainties in cost or performance. Risk management attempts to keep the process robust; it includes measurement and assessment of risk, strategies for avoiding risk, and contingency planning against risk. The manager must recognize and admit *unavoidable* risk.

An effective build process needs tool support, such as planning, budgeting and scheduling tools, effective measurement of resources used, timely assessment of progress made, support for manager and worker enquiries, and tracking of ongoing tasks and prompting for action. Support first what people spend most time doing.

Code production is the main object of the process. The work teams perform three types of task: code generation (creating new components); code adaptation (converting one component into another); and code scaffolding (building temporary structures). Component generation is the creation of new components according to a specification, with the necessary properties, of satisfactory quality, by a documented method. The same process should be followed, regardless of the method chosen to create the component. The process must therefore support all methods. Code adaptation is the revision of an existing component to create a similar component, to create an alternative component, or to create a substitute component. The process must capture and retain the properties of both the old and the new components. It may be possible later to recombine them. Scaffolding is code built as part of the process that is not part of the product. Examples are code components that create input data, inspect output data, print internal data, set up internal state, monitor state transitions, modify code dynamically, apply useful metrics, or make useful measurements.

The major tasks in building data objects are mapping values onto representations, mapping aggregates onto storage structures, and implementing verifiers and consistency checkers. The key system resources to budget are storage space and access time. Scaffolding is needed for all the above. The major tasks in building code objects are selecting and implementing the algorithms, verifying the inputs and proving the outputs, and defining the persistent internal state. The key system resources to budget are execution time and real time. Scaffolding is needed mainly for resource budgeting.

A *configuration* is a complete set of consistent components adapted to a specific target and implementing a specific requirement. The build process must recognize that it will be called upon to build several configurations, identify common parts and keep one version, identify similar parts and keep parallel versions, or highlight unique parts and keep justification. It is best to plan in advance for the likely configurations.

Integration is the combining of consistent versions of all components into configurations, with each component tested and assured. The end result is the product. There are several ways to perform integrations: bottom-up, left-right, and top-down. Bottom-up integration is an effective way to build a layered system: the lowest layer is tested and proven; the layer above is integrated and proven, assuming correctness of lower layers but assuming nothing about higher layers. This process continues layer by layer, and the final integration builds and proves the system. Left-right integration is an effective way to build a transaction processing system: the initial input is prepared accurately; the input acceptor is tested and proven; the next downstream unit is integrated and proven, assuming correctness of upstream producers but assuming nothing about downstream consumers. This processes continues along the dataflow path, and integrating the final output generator builds the system. Top-down integration is an effective way to build a command driven system: the command scenarios are generated accurately; the top level command interpreter is tested and proven assuming the lower levels exist but do nothing; the next lower level is integrated and tested assuming it receives valid commands from above. This process proceeds level by level, and integrating the primitive action routines builds the system.

System acceptance is the final assurance process that certifies the product. It determines that all components are of known provenance and quality, integration is done from a controlled configuration base, compliance with requirements is demonstrated, compliance with resource limits is proven, and the complete life history is present and well organized. Of course, the user may still be unhappy with it.

5. Coding Styles and Standards

Reading: Kernighan78

A *style* is a characteristic and recognizable manner of writing. A *standard* a set of rules that attempt to establish or enforce a specific style. The rules for a specific coding style are usually given in a document called a *style guide*. Typical stylistic features influence use of layout and white space, position and content of commentary, use of mnemonic names, grouping of code fragments into units, relationships between code units, use or avoidance of language features.

Layout is the intelligent use of white space, such as pagination between major components, line breaks between minor components, indentation to show syntactic nesting, vertical alignment of parallel constructions, logical division of lengthy constructions, and appropriate treatment of commentary. Good layout is the best aid to readability. *Commentary* is narrative explanation of code interleaved with it. There should be clear separation of code and comment and clear association of comment with code. Good commentary enhances understanding of code in that it explains its relation to other code, elucidates the purpose of code, explains difficulties of an algorithm or representation, and alerts a maintainer to potential problems. Good mnemonic names convey information such as the purpose of thing named; distinction between types, objects, and values; distinction between objects, attributes, and operations; distinction between formal and actual parameters; distinction between global and local objects; and association of generic and specific objects.

Related code fragments should be kept together; such fragments may represent attributes of one type, operations on one data store, attributes of the target machine, or parameters controlling alternative configurations. Relationships between units should be coherent, such as specific derived from generic, implementation dependent on specification, data transducers using data definitions, or higher layer using lower layer. Complex dependency graphs are a cause of confusion.

Language features affect coding style. There may be special ways to introduce mnemonic names, special ways to customize code, language-enforced grouping or dependency rules, or language constructs with useful redundancy. Style guides encourage use of such features. Language features also affect code clarity and efficiency, such as counterintuitive or hard to parse constructs, data structures with excessive space or time overhead, language primitives with unacceptable cost, or dependency structures with hidden overhead. Style guides often deprecate such features and mandate alternatives.

The overall purpose of style rules is to make the code better: more uniform, more legible, more understandable, traceable to other documents, more robust, more easily and safely modifiable, more portable, and more efficient. Uniformity is the most important reason for imposing a style. Code written by one coder is readable by another; code written this year is readable next year; units serving similar purposes look similar. This simplifies training, resource allocation, maintenance, porting, and much more. It implies that standards be consistent over time.

The *structure* of the code should be clear from its layout: what is imported, what is defined, the interfaces of closed units, the overall flow of control and its branch points, local objects and their purpose, and operations that change the values of objects. The *meaning* of code should be clear from its style: the purpose of this unit, the relation of this unit to other units, the types of all objects and values, the significance of all attributes, the effect of all operations, the algorithms used to perform computations, their domains, ranges and preconditions, and error situations and their consequences.

Style rules can help improve traceability: each unit states its purpose, each unit names what it imports and exports, each object states what *entity* it implements, each procedure states what *operation* it implements, each function states what *attribute* it implements, non-portable units state what *configuration* they support. Changes in units are tracked by a *history*.

Good style helps avoid introducing errors. Good layout guards against errors in control structures; naming conventions guard against type or role errors; use of redundancy helps the compiler find errors; good code grouping avoids unwanted dependencies; and a clean dependency graph avoids initialization errors. Experienced coders adopt robust coding styles.

Good style can make code easier to modify. It can isolate and explain configuration parameters; flag and explain temporary features; make dependencies clean, simple, and explicit; indicate where and how to enhance; and comment code with implicit properties. Few style guides consider this issue.

Good style can help make code more portable. It can isolate system and machine dependencies; specify clearly all *abstract* machines or layers; document *limits* of portability; use simple and unassuming type and object definitions; and avoid language constructs that impair portability.

Good style can help make code more efficient. It can avoid expensive language features; specify accurate ranges for types and variables; encourage clean control structures easily optimizable; avoid global dependencies, side effects, aliasing, etc.

Good style must be supported by a good process. The key is training of, and commitment by, the coders. This can be reinforced by style imposition during code creation, style checking before code release, and style improvement of existing or imported code. Some tool support is feasible. The enforcement of correct style during code creation can be accomplished with tools that allow only good code to be written, tools that reject code in bad style, prompt "buddy check" of new code fragments, style assessment by the quality assurance function or librarian, and style certification as part of release protocol. The style assessment of existing code requires reliance on code provenance or prior certification, tools that measure compliance, spot checks of code by human assessors, or full checks of code by designated sponsors.

Modification of existing code to improve style may include efforts to revise layout, improve commentary, improve structure, remove ugly features (such as "magic numbers"), regroup fragments into better units, simplify the dependency graph, and abstract system or machine dependencies. This is much harder on a large scale than on a small scale.

Several tools can impose or enforce code style, such as a language-sensitive editor, style-sensitive compiler, dependency verifier, or code improver. A language-sensitive editor is a tool for entering or modifying code that understands some of the programming language. Most editors understand only syntax; a good editor must understand also some semantics, such as the relation between declaration and use, contents of imported units, and name overload resolution. Such editors make certain errors impossible and can impose certain styles. A style-sensitive compiler is one that can be set to reject legal but ugly code. It may enforce a ban on "magic numbers," enforce consistent use of letter case or number radix, require certain proportion of commentary, require systematic use of approved alternatives (example: named parameter association in Ada), or reject attempts to use deprecated features. A dependency verifier is a tool that ensures the unit dependency graph is coherent: no circular dependencies, no dependencies of "lower" on "higher," dependencies on target machine isolated, and dependencies on data representations confined to relevant data processing algorithms. A code improver is a tool that can scan existing code to detect style violations, recommend changes, make changes automatically, or apply stylistic *metrics* such as complexity measures. Most such tools are still very simple and do very little.

6. Software Reuse: Principles

Reading: Wegner86

The main reasons for software reuse are saving time, saving cost, and being able to depend on known quality and known performance. Note that even reused software should be retested in its new environment. Saving time is the principal motivator for reuse. All software projects fall under time pressure; time saved early in the process is invaluable; an early choice to reuse saves time at every subsequent stage. However, a mistaken choice to reuse is very expensive. Saving cost is really saving *expert* effort. The skill that created the reusable component is repaid many times over during reuse; the expert is better employed on reusable components than on custom components; and the ultimate goal is to embed the expertise in the components. Known quality is an important reason for reuse because a reusable component has already been proven. This reduces the risk of developing that component, and most quality attributes are preserved by reuse: documentation, traceability, and design history. Performance is usually carried over but must be retested specifically.

Reusable components may be identified at different stages: during requirements analysis, after design, after component specification, or after component implementation. Requirements analysis may uncover a typical feature that may have one or more standard implementations; an existing implementation can be selected directly, and it then becomes a constraint on the design that it incorporate this specific component. As an example, consider a data management system requirement that calls for recovery after a system crash. One standard way to implement this is by regular checkpointing, transaction serialization, transaction logging, and rollback and replay after error. This paradigm can be adopted immediately in response to the requirement.

Reuse of design can occur when, during the design phase, a component is identified that is a reasonable part of the final system, one for which a detailed design already exists, possibly with a model implementation. The component design can be incorporated immediately into the system design. As an example, consider an application that calls for a dialogue-based user interface. A design already exists for a menu-driven interface using a standard window manager driving a command interpreter. This design can be assessed and adopted into the system. A prototype implementation could be adapted to use the specific dialogues called for.

Reuse of specification can occur when, during interface and component specification, the developer finds a specification very close to one that already exists for which proven implementations exist that can be reused and reimplemented in this system with minimal perturbation of other components. As an example, consider an application that needs to control several output devices, the device interface required is close to an existing one, and the existing interface is used in the system. Existing implementations (devices) are used as models from which to build the new implementations (devices).

Reuse of implementation is the lowest level of reuse. It can occur when, after component specification, the developer finds a specified component already exists that can be inserted into the system and will meet the requirements. As an example, consider an application that needs to sort a set of records. The language or system support services already provide a sort routine that can be invoked directly by the application. It will perform the sort according to the requirements.

Reuse occurs at several levels, and so reusability must be addressed at several stages of development. The development process should recognize software reuse as a useful practice, encourage reuse during system design and development, identify new components that might be reusable, and use and maintain a *library* of reusable components. The component library is an organized collection of reusable components that contains many types of component, each certified as reusable, and readily accessible during software development. Every development effort should aim both to produce a product and to enrich the library. Approval to write new software should be given only when reuse has been proved inappropriate.

The development process must provide *guidelines* for creating reusable components, addressing issues such as the limits of application domains, appropriate parameterizations, design and documentation standards, performance analysis procedures, and certification procedures. An essential guideline is an interface standard that states how the component must interact with others, including control and data flow, visibility restrictions, and error detection and handling. A common interface standard is increasingly a feature of advanced software environments.

Reusable components must possess certain properties: they must be usable, applicable, findable, integrable, of adequate quality, and traceable. A key process issue is how such components can be built. Software must be *usable* before it can be reusable, so a reusable component must be fully documented, fully tested, benchmarked in a typical situation, and proven in a real system. Only then is it known to be of reusable quality. For this reason, it is very hard to build a reusable first version of any component. Software must be not only usable but *useful*, so a reusable component must perform useful work required by many applications in an appropriate manner. A useful component implements a functional unit that is a characteristic of the application domain. Findability is important because if you cannot find it, it might as well not exist. Reusable components must be collected, classified, indexed appropriately, and held in a component library. Looking for reusable components should be a specific and simple part of the development process. The component library is part of the development environment. A component is more than a functional unit; it is a unit that can be integrated with other units into a complete system. The interface of a reusable component must be accurately described, appropriate for the context of use, consistent with the neighboring components, and suitable for unit and integration testing. This is made easier by a common component interface.

Quality is critically important; if it’s no good, who needs it? A reusable component must be of adequate quality. Sometimes, quality or performance requirements are so strict that a custom component is necessary. More often, the requirements are reasonable but the component fails to meet them. This is often because the builder of the component was not working to any standard of quality.

Traceability is also important; a software component must have a provenance: authors, original requirements specification, original design, test and certification procedures, and a history of relevant previous reuse attempts. The same traceability is required for reused components as for new components.

[The lecture introduces a small case study for a student exercise.]

7. Software Reuse: Practice

Reading: none

Software reuse in practice can be seen in four simple examples: a Fortran mathematical library, an Ada mathematical library, the PostScript language, and a queue manager described in the case study.

Several characteristics of the Fortran mathematical library have made it successful: its specification is almost unchanged for 30 years; it is available in all language implementations, it is controlled as part of the language standard, and it is satisfactory to most users. The standard has some disadvantages: each data type uses a different name (SIN, DSIN, CSIN), and there is only simplistic error handling. But its advantages are greater: it is integrated with language, uses typical language style, has concepts familiar to user, and has proven functionality.

In contrast, the Ada mathematical library has been under development for more than 10 years; it is still a draft specification, it has no commercially available implementation, and it is not yet a success. The main reasons for the contrast are that the Ada library is not developed and maintained as part of language, there is no quality assurance of implementations, and there has been no resolution of serious interface issues, such as subtypes and exceptions.

PostScript is a *page description language*, an interface language for programs and output devices. It was created by private venture, voluntarily adopted, supported by a growing number of programs and devices, and it is the form in which many documents are now distributed. It is likely to be a major success. Significant features include a specification that met a clear need, commercial support, an initial base of commercial products, a very clear and accurate specification suitable for machine processing, and a simple text representation that can be sent as ASCII.

The queue manager is a generic package specified in Ada. It addresses a familiar problem with standard solutions, it is designed using good object-oriented techniques and implemented as specified, and it has a test suite constructed from the specification. The component was abandoned in the second version of the product, hence reuse was not a success. The problem was apparently overgenerality: it included a complete set of operations not needed by any one user, a generalized data structure supporting all operations, and operations that were expensive to execute yet required in time-critical situations. There was no initial performance requirement, budget, or test, and the implementation was not tailorable for performance.

Reuse success seems to depend on the perceived value of the component, an accurate perception of the context of use, good integration of the component with its environment, and an initial quality product proving the component.

Lessons learned from hardware may apply to software reuse. Computer hardware is typically used for many different applications without customization or modification, and with reasonable success. This is a form of reuse. What features of hardware design make this possible? Some principles can be elucidated: design against a broad set of requirements; design from a set of guiding principles; implement simple, fast and consistent primitives; avoid special-purpose features; do not let features unused damage features used. Recognize that there will be one or more layers of design above the reusable level provided.

The reusable component should be recognizable by the application developer, occupy an obvious place in the system, conform to the expected conventions, and be part of a comprehensive set of components. A set of reusable components should be individually simple and obvious, collectively comprehensive, individually usable as required, and collectively consistent and integrable. A set of reusable components must be at a common level: they should implement (most of) a layer of abstraction, be capable of being composed into higher layers, and assume only standard lower layers. A set of mathematical routines is an obvious example.

But consider the Ada mathematical library. It uses standard exceptions to signal errors, requires a (missing) lower-level standard for error handling, and is not composable into a robust higher layer. It uses unconstrained data types, and it is not composable into a higher layer with constrained types. Because of the lack of other agreed standards, the design has to solve problems at different levels.

An application domain has its own architecture: layers that represent conceptual abstractions, interaction models that represent information transfers, objects that are common structural components, and pervasive quality or performance requirements. A *domain analysis* is an investigation of a domain. It requires study of major working products in the domain to determine the key features of the domain, identify common problems and solutions, and derive a domain architecture. This analysis, if done in the proper way, can define an architecture to support software reuse in the domain.

A *paradigm* is a common problem within a domain with a standard, proven set of solutions that can be reused in successive products. The paradigm is therefore a reuse opportunity. As an example paradigm, consider the *timeout*: two parallel processes are coupled; one is blocked awaiting the other, but it dare not block indefinitely. The paradigm solution is the timeout: the blocked process sets an upper bound on blocking time, and a watchdog guarantees to unblock the process if necessary. Design and implementation of the watchdog is a standard exercise for which several solutions exist.

A standard solution to a problem is not enough. It must be embodied in a structural component separately specifiable and buildable, independently testable and assessable, and subject to separate configuration management. Reuse of the solution then implies reuse of the component. As an example, consider a transaction log. It solves a standard problem, is implementable by a single component with a clear interface to the rest of the system, and is separately buildable and testable. As a bad example, consider a garbage collector. It solves a standard problem, but different applications impose incompatible constraints. The implementation requires detailed knowledge of the rest of the application, and it is almost impossible to generate realistic test scenarios. With luck, a high-level design may be reusable.

A *component library* is a set of reusable components of proven utility, under proper configuration control, forming comprehensive sets of objects at appropriate levels of abstraction. As an example, consider the X-windows specification. Practical implementations exist, versions are clearly differentiated and controlled, the specification provides adequate functionality for driving a large class of terminals, the specification layers are clear and consistent (one layer of data structures, one layer of routines), and the interface follows and builds on existing standards.

8. Automatic Software Generation

Reading: Martin85; Ng90 for background

Very little software is written in machine code so, in a sense, almost all software is automatically generated. The continuing goal of automatic software generation is to raise the level of abstraction of the human input. At present, we are moving from traditional programming languages to what are called fourth-generation languages. The jargon term for them is "4GL." The third-generation languages are procedural, algorithmic higher level languages such as Pascal, C, and Ada. The next-generation languages are one or more of non-procedural, not wholly textual, specification or requirements oriented, or more formal and algebraic.

Programming languages may be implemented in three ways: compilation to machine code (code generation), compilation to threaded code, and direct interpretation. Each of these techniques is applicable to 4GL. For compiled implementations, the intelligence is embedded in a compiler: the source program is converted to machine code, the machine code is constructed afresh each time, and different source programs compile to different object programs with little in common. For threaded code implementations, the source program compiles to a sequence of subroutine calls, every primitive operation in the source language is implemented by a subroutine in a component library; this library is constructed once for each machine; every subsequent program is implemented by selecting and combining the reusable subroutines; and the sequence of calls is unique to that source program. For interpreted implementations, the same component library is used as the basis. The source program is executed directly by a single interpreter, there is no persistent transformed version of the program, and the interpreter identifies each operation and then immediately calls the corresponding subroutine.

All three techniques have these common features: an algorithmic source program, an engine that analyzes the source for its operational meaning, a means by which the target machine can perform the intended operations, and a way of ensuring the machine operations are invoked appropriately. The same features are found in the next generation of languages. The main differences are that the source input is at a higher level of abstraction, the input is not algorithmic or operational, and the analysis engine must be more sophisticated. But the result must still be implemented by machine operations.

Five examples of higher languages are an input description language, an output description language, a functional language, a knowledge-based language, and a graphical application generator.

An input description language is a formal way of describing input data. From the description, a data recognizer can be built to validate input, accept legal input, transform it into appropriate internal form, and reject invalid input. An output description language is a formal way of describing an output document (its format and contents). From the description, a report generator can be built to extract the required data, convert it to the required representation, and lay it out in the required format. With a functional language, the source is a functional description of the solution; that is, a mathematical formula with a defined value. An appropriate engine can cause the formula to be evaluated, and the result is the required output of the program. A knowledge-based language has source input that is a set of rules that define necessary properties of input, invariant properties of persistent data, and required properties of output. The system must ensure the rules are obeyed, thereby processing input, generating output, and preserving information. An application generator has source in the form (input/process/output). Input and output are separately defined, and the process is defined by decomposition into subprocesses, ending with standard predefined primitives. This is converted into an application program that will execute the top-level process.

One finds 4GL in domains with persistent data, highly stylized data processing, very stable application needs, exact formulas defining output, exact rules describing system behavior, and precise graphical analogues of system operations.

In domains with persistent data, the intelligence of the system is in the data dictionary—the set of definitions of persistent objects. This is used by every application; most applications perform standard operations on the persistent objects; and the application can be generated automatically from a statement of what is to be done to whom. In domains with highly stylized data processing, almost all applications require similar operations: locate, extract, reformat, update. A small set of operations can be implemented once, and an application can be generated simply by composing the operations in the required order. Where the application domain has a body of expertise that can be codified as rules or formulas (a number is prime if ...; a book is on loan if ...), an application is simply a set of rules to be obeyed or formulas to be evaluated. There is no machine code as such; the implementation engine interprets the rules and derives operational means to satisfy them. In domains with graphical analogues, the applications typically perform processes on objects; objects and processes can be represented by icons; their association can be represented by an arc on a graph; each icon can be further elaborated by an appropriate 4GL.

Building a software generator is a difficult and costly task. We must analyze the application domain; identify objects, operations and rules; implement primitive operations as reusable components; define appropriate languages; build language interpreters; integrate the result with suitable development tools; explain how the object base or rule base is to be created; and create model object and rule bases. For example, consider an information systems application generator. It will involve a data definition language (structure of stored objects), an input definition language (data entry and validation), an output definition language (report generation), data manipulation primitives (allowed operations), a data manipulation language (composition rules), an application builder (associate input, process, output), a test generator, a model data generator, a scenario builder, etc. It needs a friendly user interface and configuration control for all of the above.

Automatic software generation has its problems: very high initial cost; techniques that are often domain specific; difficulty in estimating or guaranteeing performance; difficulty in testing the application thoroughly; and basic assumptions deeply buried in the system.

9. Software Prototyping Systems

Reading: Turner86

A *prototype* is an early version that will be discarded. A *prototyping system* is a software development system for the construction of prototypes. A *prototyping language* is a language in which prototypes are written, usually supported by a prototyping system. *Prototyping* is the technique of assisting software development by the use of prototypes.

The purpose of a prototype is to answer a question. The developer lacks certain information that is important to the development, and it can be obtained from a prototype. Typical questions a prototype might answer are: is this really what the user wants, how can we fit these components together, can we build this component, how likely is it that the system will perform as required? Naturally, different questions can imply different prototypes. A prototype can also be used for requirements refinement: build a demonstration or working model, demonstrate it to the user, and amend requirements in light of user feedback. Usually, the prototype evolves during the interaction, and the revised requirement is then reverse engineered.

As an example, consider a system in which the initial requirement calls for "a user-oriented query interface." The prototype demonstrates a "query by example" system using a dummy data base. User feedback suggests, perhaps, better layout of forms and a prompt facility to help fill in forms. The requirement can now be revised and refined.

How can existing components be integrated? A prototype can explore integration issues such as data type commonality, proper data flow, and appropriate invocation, control, and error handling. For example, the requirement may call for "reuse of the Ada mathlib." The prototype builds a typical system component making use of mathematical library, using application related data types, and implementing required data validation and error handling. If this component works, it is probable that the other components also will work; if not, the requirement must be reconsidered.

Prototyping can help in feasibility analysis. Suppose the requirement calls for something novel and hard, such as a new target machine, some other new hardware component, a new programming language, or a new software development method. The prototype demonstrates that it is feasible to introduce the novelty into the system. For example, suppose the requirement specifies a new, difficult component: "data communication shall use the new X-bus." The prototype builds a pair of bus drivers in order to verify the communication model, explore a range of protocols, demonstrate data transmission over the bus, and assess performance. If the prototype works, the component has been shown to be usable as required.

Prototyping can help in performance analysis. Suppose the requirement has strict performance goals. The components crucial to meeting the goals can be identified and isolated, prototypes of each are built, and their performance is measured using realistic scenarios. This shows whether the goals are reasonable. For example, suppose the requirement sets a bound on transaction time: "radar image must be identified within 200 ms." The prototype takes a set of typical radar images of friendly and hostile vehicles, uses several image matching algorithms, and measures the average and worst-case time of each. If one or more algorithms meets the criterion, the prototype shows that the performance can be achieved.

These examples have an obvious common factor: there is a key problem whose solution is crucial to the development, we are not sure of the solution to the problem, and a prototype can increase our information and hence reduce the uncertainty. Prototyping is hence a risk reduction process.

A prototyping system is a software base for building useful prototypes. It includes a prototyping language, component library, component simulator, scenario generator, and metrics support. The component library is a set of standard components for building prototypes. They are customizable and integrable, capable of being measured for (simulated) performance, and may include alternative algorithms where appropriate. These components are often domain-specific; the issues of software reuse also apply at the prototyping stage. The component simulator is a way to pretend components are there when they aren't; it may include stubs, "canned" value generators, input loggers, output prompters, or null transducers with artificial delays. The fake component does whatever is necessary to keep the scenario moving without loss of realism. Scenario generators simulate realistic operating conditions; they generate appropriate *sequences* of data or events corresponding to a probable history; they therefore allow reasonable overall behavior and performance to be verified and measured. Systems that maintain extensive *history* should be tested with lengthy scenarios.

It is often necessary to *measure* the performance of a prototype, so the system must support the measuring process. This often includes data store sizes and activity, queue lengths and arrival rates, and (simulated) time to perform transactions. The system must permit non-intrusive monitoring. The measures obtained must be scalable into estimates of target performance.

A prototyping language is the notation used to describe the prototype. It is written by the prototype engineer and executed in some way by the system to give an appropriate simulation of reality. A prototyping language has special features: flexibility; support for rapid *incremental* change; support for *partial* integration using novel, standard, and dummy components; and giving software artifacts that are measurable overall and in detail. Example languages include Miranda (a pure functional language with a formal bias), REFINE (a rule-based language with a graphical interface), and Proto (a graphical language with advanced tool support).

Prototyping has obvious advantages; it can help to obtain crucial information early in development, validate feasibility of key requirement, explore novel technology, demonstrate a model to the user and obtain feedback, select the best algorithm for a key component, and obtain timely performance estimates. Sometimes, only prototyping can meet these needs. However, prototyping has certain difficulties, including technical problems, process problems, and procurement problems. A technical problem is the trustworthiness of a prototype. A prototyping language is not an implementation language, the prototype components are not real components, and simulated performance cannot be directly perceived. A good prototyping technique must be quite clear about how the prototype resembles real life and how the prototype differs from real life. Remember that a prototype tries to answer a specific question. Some process problems are: the need to keep both prototype and real versions; traceability between the prototype and the real product; parallel development of requirements, design, and prototype; tension between the need for *fast* evolution and the need to keep proper development history; and the question of whether prototypes should be kept even during maintenance. Prototyping makes hard assumptions about the process: the requirement is subject to further negotiation, the user is available for lengthy and detailed interaction, adequate domain knowledge and components exist, and, most important, enough time exists for a prototyping cycle and the key risk issues can be identified and prototyped. It is sometimes hard to convince the customer of the need to spend time and effort on prototypes.

10. Software Testing: Principles

Reading: Beizer84, Myers79, Bastani85

Testing is a specific part of the software development process and serves a specific purpose. It is applied to each component, however created, to groups of components, and to the system as a whole. Any software object must pass between two stages, development and deployment. The transition between them requires *acceptance* of the object. This acceptance implies a degree of *confidence* in the object being accepted. The purpose of testing is to increase confidence in the object under test. Every test should help build confidence. Testing is complete when a satisfactory level of confidence has been built. A tested object is then *deliverable* by its developer and *acceptable* to its customer. Note that both developer and customer must have confidence in the object.

Testing is a necessary part of the process; it provides confidence about certain properties of the object that cannot be otherwise provided. Good testing therefore focuses on the properties that must be tested, the degree of confidence required, and the most effective way to achieve it.

The test process spans a large part of the development: test against specification, test against design, test of component interfaces, test of component function, integration testing, and system testing. It runs in parallel with the main development process. For each major test area, one must determine what is to be tested, what it is to be tested for, how it is to be tested, who is to perform the tests, what the actual tests are, and how to interpret and act upon their results.

Every test provides information; ideally, every possible result should provide information: a positive result implies a possible error will not occur, and a negative result implies a specific error has occurred. In either case, the test should isolate what can now be assumed acceptable and what must now be repaired. All testing is against some requirement, and these requirements are imposed at different stages. For example, in specification: "the program shall accept only valid input," in design: "input will be validated by module IVM," in the building plan: "module IVM will use a finite state machine recognizer," or in the implementation guide: "IVM will always reset itself to state NULL before exit." In each case, who delivers and to what customer?

There are three main ways to test components: black box testing, white box testing, stress testing. There are two main ways to test larger pieces: probable-case input, and worst-case input.

In black box testing, the component is opaque to the tester, the tests are generated from the specification, they are based on partitions of the input and output domain, and each test verifies correctness of part of the mapping between these domains. It is very hard so to test modules with internal state. In white box testing, the component is transparent to the tester, the tests are generated after implementation, they are based on control paths through the code, and each test demonstrates successful traversal of a specific path. It is very hard to test modules with "spaghetti code." In stress testing, the tester is trying to *break* the component. The tests emphasize probable hard cases: extremes of input and output domain, singular points or discontinuities, especially intricate control paths, and unreasonable error input.

In probable-case testing, the system is tested with typical input. If it handles the more likely cases, it will function correctly most of the time. The confidence level is therefore based on the probability of a correct response. This is analogous to the mean time between failures. In worst-case testing, the system is tested with critical input (input for which failure would have disastrous results). If the system handles the critical cases, its failure will not cause major damage. The level of confidence is therefore based on the probable loss due to system failure.

The ideal test strategy combines all these approaches and has two driving motivations: to minimize the *number* of tests (accomplished if each test excludes largest remaining error class) and to minimize the *risk* of using the delivered component (the product of failure probability and failure cost).

Testing can be performed by three groups: those building the deliverable, those accepting the deliverable, or an independent test group. In practice, all three groups should test. For example, a component might be tested thus: white box by builders, to exercise the entire component; black box by customer, to prove full functionality; and under stress by a separate group, to assure robustness. Independent testers, and diverse test methods, build more confidence in the component.

Test interpretation is important. (If you can't use the answer, don't ask the question.) A test can give several results, and the tester should decide in advance what are the correct results and what to do with each incorrect result. Badly designed tests elicit the familiar responses, "That's not a bug, it's a feature," and "Sorry, that's not my bug."

11. Software Testing: Practice

Reading: none

Testing must be considered at several stages in the development process, at several levels of abstraction, as a process extended in time, and as a process involving many different parties. The model of developer/customer is a unifying thread. Test specifications descend the levels of abstraction (requirements, specification, design, build plan, interface, code), while testing ascends the levels of abstraction (white box component test, black box component test, integration test, system test, product test, acceptance test). Each level of testing in some sense assures the entire development process beneath it.

One can regard the testing activity in two ways: as a discrete, separate stage—a *threshold*—or as something done in parallel with development. The latter leads to the notion of *incremental* testing: test as soon as able, and retest only on modification. Both approaches have their advantages. Staged testing has the following advantages: a single consistent object is tested, the tests are a complete independent check, test success is a milestone in the process, and the customer can view the testing live. It is a necessary part of the acceptance process. Incremental testing has other advantages: effort is spread over longer time, gives early feedback about defects, allows regular progress tracking, and permits refinement of tests with experience. However, earlier tests must not be invalidated by later component development.

Regression testing follows the rule “retest after change.” The purpose of a regression test is to reassure us that the change has not damaged the object. Regular regression testing is necessary during component evolution, and it is therefore an essential part of incremental testing. (It is also an essential software maintenance tool.)

Known defects can be *tracked* by specific tests: a test reminds us a defect is still there, a test assures us a defect has been cured, a test warns us if a defect reappears. Each test tracks a particular defect. Regular regression testing guards against reappearance of known defects. A defect control process ensures that all defects are tracked. The process steps are: from the error behavior determine the cause (defect), design a test specifically to demonstrate the defect, include the test in the test set at the appropriate level, generalize the test if appropriate to look for a class of defects, and repair the defect. Do not repair until you have a test to prove the repair works.

Tests are derived in three main ways: from specification, from component structure, and as stress tests. In practice, each process uses certain rules of analysis. Testing from specification uses functional partitioning, input domain partitioning, boundary value analysis, and cause/effect analysis. Testing from structure uses control flow analysis, basis path derivation, loop control determination, and data store transition analysis. Testing for stress may use both specification and structural details to identify domain extrema, determine singularities in target representations, identify size or quantity boundaries (“fire walls”), exercise implicit control paths, or violate assumptions or preconditions.

Tools support is very helpful in developing tests; for example, requirements or specification analysis tools, domain analysis tools, and code analysis tools. They can often generate the majority of the tests that are required in practice. Specification analysis tools work from a formal specification of the product to identify functional areas, determine inputs and outputs, generate test data and model results, and flag areas of excessive complexity. This works best if the various functions can be cleanly separated and described. Domain analysis tools work on a description of the application domain (objects, attributes, transformations, invariants). They generate tests to determine attributes are correct, transformations work, and invariants are preserved. They are usually driven by a rule-based domain description. Code analysis tools use the structure of the code to generate tests, build a flow graph, generate basic path exercisers, generate loop iteration and termination tests, test exceptions or other implicit control actions, and check coverage of input domain. This analysis can generate proofs as well as tests.

Test-processing tools automate the process of testing: when to test, what to test, what has been tested, and what are the results of the tests. For each object, the tool maintains the current set of tests with provenance of each, a test versus requirements map, a test coverage map, a test result set and appropriate actions, and a current defect list. This is keyed to the object version and configuration. The test harness performs the tests: select a test subset, execute tests and log the results, compare the results with stored result sets, modify the defect list as appropriate, and recognize anomalies and alert the appropriate manager.

12. Meeting Performance Constraints

Reading: Graham77, Sha89

Performance requirements are normally expressed in three ways: response time, throughput, and flexibility. They also give some indication of normal workload and abnormal workload. Examples include response time: “not more than 10 ms”; throughput: “at least 100 per second” (note that these are not equivalent); flexibility: “must be able to handle 150% normal load for up to 2 seconds”; normal workload: “not more than 100 per second”; abnormal workload: “up to 300 in 2 seconds.”

The software design will have defined paths followed by each transaction, resources required by each transaction, arrival points of transactions, and internal buffers or queues. The overall performance requirements can then be tied to specific components or sequences of components. The simplest approach is to *budget* for performance: each component is given a time budget and must be implemented within that budget; the sum over the transaction path must be within the requirement. If a component participates in many transactions, its budget is the maximum allowable for the *most urgent* transaction.

A simple budget process has these uncertainties: length of time to perform the operation, arrival times of transactions, length of time waiting in queue, and length of time waiting for shared resources. The time to perform an operation is difficult to determine because few algorithms can be timed exactly, and execution time is often data-dependent. Analysis can determine the best case, normal case, and worst case. Budgeting for worst case is usually too costly. The normal case is that most likely to occur, but it is not the average case for several reasons: worst case can be very bad (even infinite), one slow transaction can delay later ones, and worst case instances come in clusters. The last problem is a common pitfall.

Transactions arrive from the outside world at random, but there is usually some underlying distribution amenable to formal analysis, such as mean time between arrivals and the probability of N simultaneous arrivals. There is a calculable trade-off between processing time and probable waiting time. Transactions within the system do not arrive at random, they cannot be generated faster than the producer can run, and they will arrive in bursts when the producer flushes its queue. In general, this rule applies: if consumers can keep up with producers, there will be no internal delays. The system can then be treated as a black box with a single queue at the front. If all transactions are identical, queueing time can be estimated accurately. Several queue models exist with known properties: FIFO, round-robin time sliced, and priority based. If transactions are different, the problem is more complex.

Transactions may have to wait for resources, such as free buffers, access to shared data structures, or specific hardware components. This is especially a problem with common resources since any transaction may then block any other. Shortages can be reduced by permanent allocation of resources to transaction types, preemption of resources by priority transactions, adaptive monitoring of resource usage, and load balancing. Usually, the difficulty is not in taking corrective action but in knowing which resource is scarce. In some cases, it is possible to preallocate all resources, including storage, processing components, and processing time. If arrival and processing times are known, then overall performance can be guaranteed.

An overload is a load in excess of design or estimate. There are two key problems: handling a transient overload and returning to normal after an overload. There are two alternative strategies: load *shedding* and load *deferral*. Each solves one problem at the expense of the other.

There are several tools for estimating performance, including transaction cost and frequency estimation, transaction cost summation, queueing theory, resource estimation, and scheduling theory. Finally, performance can be estimated by modelling and simulation.

Performance must always be measured to demonstrate to the user that the requirement is met, to confirm prediction or estimation, or to provide data for performance improvement. Typically, specific improvement action is needed to meet reasonable performance goals. These measurements provide useful data: transaction arrival times, transaction waiting times at each handover point, transaction processing time (both overall and in detail, component by component), queue lengths and length distribution over time, and recovery time after overload. Measurement tools include instrumentation (special code inserted into components to record data), monitors (special devices attached to the system that capture data), and analyzers (code or devices that reduce data to useful information).

Performance improvement is usually based on the premise that most systems exhibit hot spots (specific components that consume most of the time) and bottlenecks (specific places where transactions wait in bunches). If these can be found, a *small* and *local* change may yield a dramatic *overall* improvement in performance. Code tuning is an iterative process: find the component taking the most time, extract it for revision, tune it until satisfied with the new performance, replace the tuned component, find the next hot spot. This is repeated until the performance goal has been met or no further hot spots can be found. Bottleneck deletion is also an iterative process: find the worst bottleneck, determine the reason for the blockage (such as insufficiency of one resource, inadequate priority of consumer, or inadequate performance of consumer), and repair as appropriate. Repeat until no bottlenecks remain.

13. Implementation Metrics and Costing

Reading: Boehm81, McCabe76

A cost-effective build process must estimate costs at the start, allocate resources as estimated, monitor actual costs and progress, detect potential problems, and take corrective action. This requires good estimates, good monitoring, and sufficient flexibility in the process.

A *metric* is a standard for measuring something, such as the size of a software component or the complexity of a software product. A *cost* is a quantifiable input needed to make something, such as time, staff, or money. A good metric allows one to *predict* probable costs.

Various measures have been proposed for the size of a software product, but the one that seems to yield the best metric is *lines of code* (LOC). This provides a standard of comparison for products in the same application domain written in the same language using the same style and process. Novelty in any of the above creates uncertainty. The most common cost measure is *man-months* (MM). A given organization can usually convert this into money. Over a narrow range, staff levels can be traded for time: we can estimate time to deliver with given staff level and can estimate staff level needed for given schedule.

A *cost equation* explicitly relates an attribute of a product with the probable cost of producing it. For example, if we measure size in thousand lines of code (KLOC) and cost in man-months (MM), then a sample cost equation is $MM = 3.0 * KLOC^{1.12}$ [Boehm, Ch. 8, table 8-1]. For example, suppose we estimate a product to be 100 KLOC. The basic cost equation tells us it will take us $3.0 * 100^{1.12} = 520$ man-months. This is unrealistic for two reasons: neither LOC nor MM is precisely defined, and the equation takes no account of team skills, problem complexity, availability of tool support, etc.

A *cost driver* is a quantifiable factor that affects cost, and may be an attribute of software product, target platform, building team, or build process. Estimated values of the attribute are converted into *scaling factors* by which the base cost is multiplied. Product attributes are those that affect its cost, such as the required reliability or perceived complexity. Target attributes are features of the target machine, such as execution time constraints, memory constraints, or platform volatility (likelihood target will change). Team skill cost drivers reflect the skills of the team in the given domain and with the given development tools, including domain experience, target machine experience, and language experience. Build process attributes include use of managed process and use of tools.

For example, suppose the 100 KLOC product has the following features: very high complexity (1.30 scale factor), interactive development system (0.87), team having limited language experience (1.07), and only minimal software tools used (1.24). These cost factors multiply to yield 1.50. The weighted estimate is therefore 50% greater than nominal, or 780 man-months.

The basic size measure, lines of code, must be estimated before cost can be predicted. It is usually hard to get a good estimate before detailed design and build plans have been produced. Typically, large components are underestimated. Accordingly, estimates grow as components are refined and subcomponents are identified. Early estimates should be based on past projects.

The other cost drivers are also hard to estimate. Most of them imply the organization has a body of past experience against which to compare this project. The reference gives some guidelines how to estimate. Note that the final result is very sensitive to some drivers, especially product complexity.

Since a cost estimate depends on many factors, those factors can be changed to change the cost. Some of these factors may be under project control, such as the skills of the implementation team, programming language, and implementation platform.

Tracking costs implies knowing at any point how much have we spent and what have we spent it on. Each cost must be charged to a process or product unit: “cost of testing storage allocator is 2 MM to date.” The units to which costs are charged must be the units by which progress is measured.

It is harder to measure progress than cost. The best way to get a *quantitative* measure is to have a large number of units and have a firm yes/no criterion for each unit: “storage allocator fully unit tested and installed in library.” Each unit represents one clear stage in the development of one definite component or subcomponent. Costs are charged incrementally up the integration tree.

The first sign of a problem is a divergence between an estimate and actuality. Accurate and *prompt* measurement is needed. A cost overrun implies a *false estimate*; the estimate must be revised to yield a new cost and any related estimates must also be revised. It is not enough just to “eat” the cost and push on. Revising estimates is done by determining the cost driver whose value is wrong, deciding on a better value, examining other related components for which the same change might be appropriate, rerunning the estimating process with new values, and examining new cost estimates for sanity. This might require more than one iteration.

The test of a good process is its response to problems. Several kinds of response can be taken: rebudget and press on regardless, apply more (or more skilled) effort, reduce the scale of the product, or improve the technology base. Some responses are short term, some are longer term. Stretching the schedule is almost an instinctive response; the solution is to slip the schedule and keep working. However, this is probably the *worst* response. The manager is under pressure to make the minimum slip, and the consequences of the slip are also underestimated. The product delivery date therefore recedes gradually into the future and the team become demoralized. This is another instinctive response: the deadline is held by throwing effort at the problem. This usually fails because adding new people degrades the skill level of the team (a typical degradation in application experience and programmer skills increases cost by 30%). Only *more skilled* people should ever be added to a team. Reducing functionality is often considered a last resort response. In effect, parts of the requirement are deferred. This is one of the better responses, since it is effective and it will probably happen anyway. This of course requires negotiation with the customer. Reducing constraints is the best short-term response. Negotiate away expensive and irrelevant constraints, such as a complex and unfamiliar programming language, inadequate and obsolete deployment platform, real-time requirements unrelated to reality, or excessive volatility in requirements or platform. These constraints are usually a result of overspecification. Unfortunately, the procurement process often encourages this folly.

A longer term response to habitual failure is to improve the software development process and its associated technology. This includes training people, evaluating and installing good software development tools, improving the project management process, and improving cost estimation and budget planning. Improve in this order: processes, then people, then things.

Some projects prosper better than others. However, any organization has a *base level* of competence. This is sometimes called the *process maturity level*. This level can be determined by self-assessment and analysis of sample projects. An organization can become more cost-effective by raising its maturity level. Each maturity level has specific characteristics, such as use of advanced tools, defined and imposed quality assurance as an independent process, and regular staff training programs. An organization advances from one level to the next by instituting the *appropriate* change in its process. It is important to avoid *inappropriate* change, such as introducing tools without proper training or attempting cost control without proper metrics.

14. Midterm Review

15. Midterm Examination

16. Software Maintenance Overview

Reading: Glass81, Holbrook87

The purpose of software maintenance is to preserve the value of software over time. Anything done to a deployed software artifact with that purpose is maintenance. Maintenance ends only when the artifact is obsolete.

In order to preserve the value of software, we must know what gives software value, what enhances its value, and what detracts from its value. Appropriate actions then need to be taken, and appropriate processes instituted. Software is valuable because it meets a genuine customer need, it is easy to understand and use, it makes efficient use of resources, and it incorporates appropriate up-to-date technology. Accordingly, software can enhance its value by expanding the customer base, meeting additional requirements, becoming easier to use, becoming more efficient, or employing newer technology. Factors that reduce the value of software include errors, contraction of the customer base, changes in customer requirements, problems with understanding and use, inefficiencies, and obsolescence of system technology. Value reduction arises from several sources, including intrinsic errors, unfriendliness, and inefficiency; changes over time in customer and technology; and changes in our paradigms—our view of how software should behave.

Maintenance is considered to fall into three classes: corrective (repairing intrinsic defects), adaptive (responding to changed circumstances), and perfective (improving working software). Pressman estimates that 50% of all maintenance is perfective, 25% adaptive, and only 21% corrective. (Other miscellaneous reasons account for the last 4%.)

Effective software maintenance is difficult. It requires gathering information that identifies a need to change, deciding what changes to make, and deciding when to make a change. Maintenance must preserve software quality under change. It is also difficult to deploy maintenance resources effectively and to propagate changes through the installed base.

Information implying a need for change has many sources: existing customers, potential customers, market research, competitors' products, or technical strategic planning. The organization must often actively seek such information. There are three factors driving change planning: technical issues (what changes should be made), marketing issues (when to make changes), and budgeting issues (what changes are affordable). Any changes must be feasible, cost-effective, and timely. The organization should instigate changes for planned, timely introduction. Software changes are normally incorporated in *versions* (new releases that replace old ones), *configurations* (additional releases alongside old ones), and *patches* (small changes to installed versions). In an ideal world, all configurations exist in the same version and there are no patches.

Managing software change involves these principles: keep an accurate history of each artifact, introduce new versions effectively, plan new configurations appropriately, preserve all necessary expertise in-house, track the cost of each release and the income it yields, recognize a market opportunity and respond to it, and know when to retire a version, configuration, or product.

The cost of maintenance depends critically on how well the artifact has been built, the number of initial defects, the seriousness of defects, the degree of independence of components, the degree of abstraction from specific target features, the adaptability of the system to incremental requirements change, and the extent to which new technology can be accommodated. There are specific techniques for achieving these goals. Unfortunately, many organizations maintain software they did not build and over which they had no control of initial quality. They have no real awareness of customer requirements and have insufficient in-house expertise. The software may have an incomplete or missing development history and inadequate supporting documentation. The maintenance process must address these problems.

Tool support for maintenance is almost essential, including a well-organized information base; support for version and configuration control; support for testing and quality assurance processes, tracking costs and benefits, tracking persistent defects, and recording and estimating market size and demand.

A successful product resembles a successful life form in that it exhibits initial novelty, a small initial population, increasing complexity, increasing ability to compete, a dispersal into diverse environments, adaptation to new circumstances, radiation into many related species, and gradual change into unrecognizable forms. This analogy is captured by the term *software evolution*. In the world where software is built for profit, a product will evolve until the marginal cost of change exceeds the marginal benefit, a product will expand into new environments as long as the new customer base will pay for the new configuration, and a product will persist until a decisively better product displaces it. Effective maintenance recognizes these pressures and supports products for long life and wide distribution.

Maintenance costs gradually increase over time as the product eventually reaches its limit of adaptability and its old environment gradually erodes. The end result is analogous to extinction. There is an optimal time to replace an old product, and a good maintenance process will estimate that time. The organization should then plan such a replacement.

17. Causes of Software Evolution

Reading: Boehm88

Almost all deployed software changes over time, and the changes all have underlying reasons. Some change is in response to direct motivators; other change is a consequence of earlier change. Good management accepts change as necessary and plans for it.

Evolution is an accumulation of small changes, such as changes to individual components or subcomponents, addition or removal of components, or regrouping of components. Sometimes the small changes are part of a larger design, and sometimes they are made in direct response to events. For example, consider a plan to port a product to a new platform. This is a larger design that generates many small changes; when all the changes are done, the port is done. Consider now a corrected version of a product. This is simply an accumulation of small changes, and each change is in response to a separate event.

Behind every change there is some key reason, such as cost that can be avoided, benefit that will be incurred, or an identifiable source of cost or benefit. This key reason is the change motivation. It usually has both economic and technical components. Changes in software can reduce the cost of demonstrating or marketing, the cost of selling and installing, the cost of customer dissatisfaction/bad reputation, the cost of defect correction, the cost of enhancing, and the cost of porting. Costs can be divided into three types: cost to customer, cost of marketing, and cost of maintenance. Accurate estimation of these costs requires good feedback mechanisms, both internal and external.

Change can reap these benefits: customer satisfaction and repeat business, increased market opportunities, increased market penetration, and opportunities to sell other products.

Any specific change can be considered corrective, adaptive, or perfective. However, often one change creates the need for another, and changes typically come in sequences because the product enters a period of instability that ends when all consequential changes are done.

Most major changes in a product are planned, because market research finds an opportunity, a technical study demonstrates feasibility, the business group finds the cost/benefit ratio satisfactory, and management agrees to implement the change. The product then enters a new evolutionary stage. Some key reasons to plan a major product change are to meet an evolved customer requirement, to make use of novel technology, or to invade and capture a new market. As always, the estimated benefit must exceed the cost.

Evolution because of changed requirements usually has these stages: determine new or changed requirements (consult), construct a revised specification (analyze), generate a revised design (modify), determine which components need not change (salvage), implement new components (build), reintegrate, retest, reassure quality and performance (tune), and deliver to the customer. Evolution because of new technology has these stages: investigate promising new hardware (assess), determine how components can make use of it (redesign), demonstrate feasibility of a new subsystem (prototype), implement new components (build), reintegrate the product, retest, assure quality, etc. (tune), prepare demonstration and sales material (package), and release the product (announce). Evolution to capture a new market has these stages: adapt to a new platform (port), adapt to new customer expectations (enhance), maintain robustness and overall quality (restructure), ensure adequate performance (tune), provide appropriate demonstrations and publicity (package), and release the product (announce).

Most small changes to a product are unplanned: the customer encounters a problem, the technical support group finds solution, implementing the solution requires a change or patch, the sales support group insists a rapid fix is vital, and management approves implementation and release. The product now has an unplanned interim version. Unplanned changes are disruptive because they damage development schedules, generate unwanted versions, and probably reduce overall quality. These changes should always be reconsidered later, both the reason for the change and the change made. The result is either to formally accept the change into the product and propagate, or to reject the change and plan a new solution for next version. In any event, the unplanned version should be retired.

18. Management of Software Evolution

Reading: none

If most software products change over time, that change should be planned beforehand, controlled while in progress, and understood afterwards. Change management requires appropriate structures, processes, and strategies. Managers must appreciate reasons for change, key motivators, costs and benefits, change planning to maximize benefit, and timely introduction of change. Change must be tightly controlled to keep costs within limits, change only what needs to be changed, minimize unwanted consequences of change, and ensure changed products return to stability. Past experience is a guide to management; this includes the life cycle history of products, evolution of components and versions, growth and decline of markets, trends and fashions, and symptoms of specific problems.

Adequate management of software evolution requires an extensive base of information, an overall strategy, periodic consideration of tactical options, an effective means of implementation, and systematic collection of feedback. This requires support structures and processes. The management must have structures to organize information about software objects; resources, costs and benefits; products; and customers, present and prospective. Maintaining this information is simple but costly; making effective use of it is difficult.

Components, versions, and configurations are the basic units of software about which information is kept, including the life cycle of each object, use of components by products, relationships between parallel configurations, and differences between successive versions. The goal is full traceability from any important fact to the basic reason for that fact.

The information base collects historical information—specific data about components and products. This can be consolidated into statistical information—summaries, averages, and correlations. One can then generate predictive information—trends, projections, and extrapolations. Information is maintained in the support structures; it is gathered from customers and market research, assigned to the appropriate software object, and analyzed for its implications. Plans are drawn up in response for products that need special action or opportunities that need consideration. This also helps identify products that should be left alone.

There are effective ways of using the information base for managing versions and configurations, tracking and improving components, tracking and changing products, and estimating consequences of possible actions. These processes are the tactics of software marketing.

Version control is one process for managing software change. Anything a customer has, the organization has; the organization knows what each customer has; all differences between versions are documented; and there is a clear upgrade path from any installed version to the latest version. The ideal is that there should be only one installed version and that it should change infrequently and smoothly.

Configuration planning requires that every configuration targets a known environment, every specific component has a compelling rationale, common components exist in (ideally) one version, costs and benefits are maintained for each configuration, and new environments can be targeted appropriately. The ideal is that every new configuration should yield more net benefit than any feasible alternative.

Component tracking is an incremental route to quality and profit. Information is kept about each component: extent of use, maintenance cost, defect rate, and estimated replacement cost. The worst components are targeted for replacement, and the best components are targeted for reuse. Appropriate lessons are drawn about the software process.

Product tracking is one basic guide to product development tactics. The organization maintains, for each product: historical costs and benefits, projected costs and benefits, market penetration, common positive and negative comments, and present and projected “product differentiation” features. This allows one to plan development of existing products and introduction of new ones.

In summary, these are the basic principles for software maintenance management: software products are built for profit, change is needed to preserve their profitability, any change must serve a useful purpose, changes must preserve software quality and utility, and decisions must be based on adequate information.

The evolution of software products must be guided by an overall strategy based on organizational objectives and marketing objectives. Two observations are relevant: an organization is defined by its current products, and an organization defines itself by its future products. An organization may decide to change for several reasons: to become expert in a new domain, to lead in the use of new technology, to introduce software into new markets, and to become standard bearers of new paradigms. New and revised products are a way both to implement the change and to fund it. Most organizations seek to expand, to find new markets and to increase penetration of existing markets. This implies evolutionary change to adapt products to a new environment, improve existing products, or increase product differentiation. The choice of which path to pursue is a strategic decision.

19. Maintenance Tools and Environments

Reading: Grady87, Ince85

A maintenance environment is a collection of *integrated* tools and *organized* information to support software after its deployment. It is similar to a software development environment; ideally the same environment should be used for both development and maintenance. The maintenance environment has the basic purpose to facilitate all maintenance of the product during its life. To do this, it must collect information, organize information, control versions and configurations, support version evolution, and support configuration planning and evolution.

The environment must gather technical information (new versions and configurations, changes in all documents, change requests and the responses) and management information (resource utilization and consumption, sales and profits, and data on customers, markets, and competitors). The information collected must be organized: associated with the appropriate version and configuration, associated with the appropriate stage of life cycle, and tagged with importance or urgency. All consequent actions must be documented and a proper record appended. When a product reaches stability again, a complete set of data can be finally checked and closed.

Naturally, the environment must provide version and configuration control: every entity must belong to some version and configuration, version and configuration information must be known for all customers, established change processes must be enforced, and external preconditions of all versions and configurations must be maintained. The environment must recognize that versions evolve and therefore must track change requests and defect reports, identify what should or should not be changed, and maintain change rates and costs for all components. It must ensure that the proper process is followed: that schedules are set and maintained, all relevant documents are upgraded, testing and quality assurance are accomplished prior to release, and superseded versions are replaced in a timely manner. The environment must also support configuration planning: maintain common components properly, identify and maintain configuration-specific components, maintain the rationale for all differences in configurations, isolate or propagate defect reports as appropriate, and maintain correct cost and sales information.

The environment clearly needs good general tools, including a data base, data capture or entry tools, a data manipulation language, spreadsheets and financial tools, and statistical analysis tools. It also needs tools specific to its task, including technical tools that assist the process of maintenance and management tools that assist control and planning. Examples include tools to find and maintain information, such as browsers and annotators, and tools to identify where changes should be made, such as defect locators, requirements tracers, change isolators, change propagators, and consistency checkers.

A browser allows one to *navigate* the information base: to move from module to related module, go back from code to design to requirement, access and review rejected design alternatives, find changes from previous version and their reasons, and cross to the corresponding module in a related configuration. The maintainer should be able to find all *relevant* information using the browser. An annotator allows one to augment the information base. Regular data collection adds new complete entities; annotation adds new information to existing entities. It is intended to capture anything important that occurs during the maintenance processes.

A defect locator finds where a defect probably is. This can be done positively (construct a test that shows the defect and then find the components used by that test) or negatively (successively eliminate components that cannot be the cause of the defect). A requirements tracer helps implement a requirements change by determining which components are affected. This can be done forwards by tracing from requirement through design to code, but it is often more easily done backwards by determining which behavior should change, finding all tests that demonstrate that behavior, and determining which components are used by those tests. A change isolator determines the limits of any change; it identifies components that will remain unaffected and associated documents that will not change. The method is to determine the nature of the change, the primary affected entity, and all associated entities that might be affected by a change of that kind. This determines the maximum extent of the change. A change propagator ensures that a change in one version and configuration is correctly propagated to other versions and configurations. Changes in common modules cause it to generate associated change requests in other configurations, rebuild and retest, and prompt the manager for a release schedule. Changes in configuration-specific modules prompt the support group to check for similar defects. A consistency checker ensures that each version or configuration is consistent: that all documents have the correct version and configuration number, all change requests are closed or deferred, and all costs are computed and billed. They can go some way to ensuring the contents of the various documents are in accord. However, the only practical way to do this is to generate the documents from a single source.

The environment can preserve sanity by providing easy and effective capture of new data, effortless browsing and annotation of existing data, trustworthy prompting for needed data or actions, a single reference version of any significant information, and automatic generation of as much as possible.

Management tools include those to determine the need for change, such as defect trackers and change request trackers, and those to allow one to plan change, such as cost, resource and sales analyzers, scenario analyzers, and trend identifiers and anomaly finders.

A defect tracker computes the defect rate of components, versions and configurations. They allow one to identify abnormally bad components, abnormally troublesome versions, or exceptionally difficult configurations. This permits selective remedial action. Note that management should also be interested in abnormally defect-free entities! A change request tracker measures the stability of the software components: which ones change most frequently in response to users and which ones remain unchanged over many versions. They also measure which kinds of change are costly: those that involve many components or generate many defects, and those that require market research or prototyping. This information allows one to plan software restructuring. Cost and sales analyzers compute the profitability of each product: the cost to build and maintain, the gross and net profit from sales, the change in profitability between versions, and the comparative profitability across configurations. This allows management to determine which products to retire and which products to market more intensively. Scenario analyzers simulate products that do not exist in order to determine predicted costs to build and market, the rate of market penetration, the expected defect rate and cost to maintain, and the expected change request rate and version frequency. They use historical data to estimate future data. They help management make an informed choice of tactical options. Trend analyzers determine trends in temporal data and anomalous departures from the trend. This alerts management to the need to reconsider current plans in the light of changed circumstances.

The key to keeping management control is to collect and analyze all information automatically, filter all irrelevant information, prompt managers only with information requiring action, and prompt them only when it is the appropriate time for action. This is called *management by exception*.

20. Dealing with Errors in Software

Reading: none

Management must expect defects and plan to deal with them adequately; it must minimize customer cost and irritation, minimize repair and reissue cost, schedule repairs in a timely manner, and be prepared for emergencies. The organization should be able to respond promptly to customers, locate defects rapidly, decide quickly what repair action to take, and cope with the consequences of defects and repairs.

Good defect processing requires a standard way for the customer to communicate the fact of a defect and all necessary information, an in-house copy of the product by means of which the defect can be replicated or simulated, a standard way of incorporating corrections into scheduled releases, and a means of performing and issuing emergency repairs or patches. For every product, there must be a *maintenance cell* that retains all product history and documentation, keeps accurate records of releases and customers, has access to all necessary human expertise, and interacts with product and version planners. This cell is the main agent in repairing the product. Its cost is part of the ongoing cost of the product.

As well as dealing with individual defects, management should have *defect control* procedures for tracking the number and cost of defects, reducing abnormal defect rates, reducing average defect repair costs, and making important products more robust. This is part of longer-term *perfective* maintenance.

There are four key stages to correcting a defect: understanding the defect, locating the defect, implementing the correction, and reverifying the corrected objects. A symptom of a defect is that a software product has behaved in a way the user did not expect, under a specific set of circumstances. The maintainer must understand two things: what actually occurred, and what ought to have occurred. The first step is to *replicate* the defective behavior: determine which product is involved, determine the behavior observed by the user, reproduce all relevant conditions, and execute the product and observe the same behavior. If the maintenance organization cannot replicate the behavior, it should at least try to *simulate* it. As a last resort, some execution *trace* can be used ("'software flight recorder"). The second step is to decide what *should* happen: what behavior did the *user* expect, and what behavior does the documentation *lead* one to expect. In any event, there is some error, either a genuine defect in the product or a defect in the *explanation* of the product.

In general, a defect will appear in several related places; for example, an error in a module specification affects the specification document, module definition code, module implementation code, module unit tests, and documentation of much of the above. The maintainer must find all these places and then identify the *primary* occurrence. There are three steps: finding the first evidence of the defect's cause, finding all related evidence, and determining the primary cause. The first is by far the hardest and is all that is commonly understood as "bug finding." The other steps depend on good *traceability*. Bug finding has been documented at great length; the essential principles are finding suspect components, eliminating innocent components, isolating the culprit, and reconstructing the crime. The process usually forms hypotheses about the probable causes, tests them, and eliminates them. Reconstructing the defect is a key step too often omitted. To locate a defect, one must find an error, determine that correcting the error will repair the defect, and prove that the error caused exactly the defect observed. Only after this last step can one be confident that the repair will be a complete one.

Implementing a correction is a lengthy but straightforward process: repair the primary cause of the defect, construct a test that demonstrates the repair, derive the consequent changes in other objects, derive appropriate tests for each of them, and rebuild and retest upward from the changed objects. Finally, rerun the original demonstration tests and show that the defective behavior has been corrected. Most defects have an identifiable *primary* cause, which is usually the erroneous object furthest upstream; earlier objects are correct and errors in later objects are effects, not causes. The primary object must be corrected so as to be consistent with the upstream objects. It must be demonstrated that the correction is a repair by means of a specific test. All other erroneous objects must be changed. Ideally, this change is implemented by rederiving the new object from a corrected upstream object. If necessary, an object can be directly repaired. A proper consistency check is then essential to determine that this change is consistent with the primary change and this change can be traced back to the primary change. Similarly, new tests should be derived or created. Reverifying the product proves two things: that all necessary repair has been done, and that the repair has not caused accidental damage. The two processes that prove this are consistency checking and regression testing. It is here again that good traceability is a great help.

Any product evolves naturally as circumstances change, which leads to a succession of scheduled versions. Ideally, defect repair should be a part of this process, where each version is issued as scheduled and each version repairs all known defects. This allows both maintainers and customers to plan appropriately. In anticipation of a coming release, it is desirable to set a cutoff date for defect reports, consolidate defect reports, decide on component replacements (if any), consolidate repairs, integrate defect tests into product tests, and ensure that the version guide mentions all corrected defects.

Often, many repairs have touched one object, so it is desirable to review all these repairs together to look for possible underlying causes, ensure the repairs have not affected each other, and ensure that the repair has maintained overall quality. In bad cases, the component should be rewritten.

A *patch* is a repair issued in a special release of the product as a response to an emergency—a defect whose prompt repair is vital to the user. Ideally, patches never occur. If there are too many of them, the product is insufficiently robust and should be revised. The steps in the patching process are to determine that there is an emergency, authorize emergency repair, repair and retest as fast as possible, issue a release to complaining customer, notify all other customers with the defective product, and schedule a normal repair of the same defect. All patches must eventually become proper repairs. Patches should also be tracked against components, customers, and types of defect. This is useful data for product improvement. Note that the need for patches can be reduced by reducing either defect rate or user cost.

The key concept of robust software is to minimize the cost to the user of any defect. This implies two things: anticipate that defects will be present, and take precautions against their consequences. These precautions are then built into the product. We are familiar with hardware robustness: error detection logic, error correction codes, and redundancy; similar techniques apply in software: postcondition checks, invariant checks, data redundancy, and recovery blocks.

21. Requirements Evolution

Reading: Poole77

A software product is designed to a *requirement*. However, requirements change over time. If the product is to remain of value, it must change along with the requirement. This change is a part of product maintenance. It might be planned or unplanned. Often, we know that change will occur, but we don't know exactly what changes.

The customer is motivated to change. This creates a changed requirement, and we must identify the change, analyze the change, determine the impact on the product, plan the appropriate product change, implement the change, and deploy the upgraded product. There are many reasons a requirement might change: a change in external circumstances, a change in internal procedures, a change of scale or scope, a change in the role of the computer system, or evolution of a new paradigm for computer use. The consequential product changes are diverse.

If a software product is to work in the real world, it must accurately reflect part of that world. Hence, a change in real-world circumstances forces a change in the user requirements. This change must be propagated into the product. In an extreme case, the product loses all value until the change can be implemented. Procedural changes in an organization can also create new requirements. The overall function of the organization is the same, and external circumstances have not changed, but the way the organization works has changed. This implies a change in the way the product supports the organization. Accordingly, the product must evolve. In other situations, the overall requirements of the organization are the same, but that part of the requirement met by the software product has changed. What usually happens is that a computer system, once installed, is called upon to do more and more things. This causes continuous elaboration of its requirements.

Sometimes, requirements remain functionally the same but the scale changes. Over time, the system processes more information, keeps a larger data base, or requires answers more quickly. However, a change of scale in the requirement may imply a major change in software. In other cases, a change in requirement is a consequence of a change in the way we view the system. The functional requirement is the same, but the way the requirement is met has changed, such as a new view of how the system should work or a new concept of how user and system interact. The result is a change in the behavior of the system.

The first stage in implementing a change is to determine its impact, or analyzing the change for its effect on the software product. Typically, one finds three types of impact: evolutionary and planned, evolutionary but unplanned, and revolutionary. With evolutionary change, the fundamental design remains unchanged, the basic partitioning of the system remains sound, details of some modules need to be revised, and possibly new modules need to be added. But they are very similar to existing modules and use the same data and control interfaces. With revolutionary change, the change in requirement might be quite small, but it implies a fundamental change in any of the basic system design, functional partitioning, the system external interface, or internal data or control interfaces. In consequence, the system will need to be substantially revised or rewritten.

The key issues in implementing change are recognizing revolutionary change and implementing evolutionary change. A revolutionary change requires system redesign, and code can perhaps be salvaged later. Evolutionary change requires system upgrade—change occurs within existing structures.

Upgrade planning should try to maintain a balance of upgrade costs and benefits: set limits on the rate of product change, control costs of implementing change, and prioritize changes by cost/benefit ratio. Note that advance planning can reduce upgrade costs. In practice, products tend to change at the maximum affordable rate (but the market always wants more change). There are some strategies that make change easier; for example, if several changes affect each other, do them together.

Software design can anticipate change: expect objects to become richer over time, expect the command set to evolve, design against anticipated future size or scale, and identify and parameterize volatile features. Good design also documents the expected limits of change. There is a limit to the flexibility of any design: basic necessary assumptions that must be fixed, maximum size or scale that the system can handle, and the fundamental paradigm of work and user interaction. There is also a limit to the justifiable cost of flexibility—the simpler and less flexible system is a lot cheaper.

22. Technology Evolution: Principles

Reading: none

Rapid technological change is a feature of this field. These changes affect the way people use computers, the size of the problem a computer can solve, the cost of a computer system, and the way software interacts with the machine. A product must be prepared to cope with these changes. There are at least three important types of change: increase in machine power, introduction of new machines, and changes in interface technology.

Increase in machine power has two consequences: a machine of the same price becomes much faster, and a machine with the same speed becomes much cheaper. Software for the machine must evolve to take advantage of the speed increase and to take advantage of the new market for a cheaper machine. Increased speed creates a need for software revision; for example, a batch program becomes fast enough for online use, the processor can cope with a more advanced terminal, or processor power is available for background tasks. The user probably wants more than "the same but faster." A decrease in cost can open up a different market, such as individual workstations for all staff, personal computers for the home, and small computers in schools. The software may again need revision so that the new potential customers are happy with it.

The product developer should anticipate an increase in machine power. Planning includes deciding what software or hardware component determines the overall speed of the product, whether the product can take advantage of an improved machine, whether some components need to be revised, or what the product would do differently if it were faster. Adaptive maintenance is needed to cope with the problem.

New types of machine are introduced quite frequently, and these represent new market opportunities. The software must be revised to run on the new machine; this is traditionally called *porting*. Software that is easy to move is called *portable.* Software depends totally on the machine beneath it. Nevertheless, it can be made portable—less dependent on a specific machine, insensitive to changes in machine components, and structured to make revision easy. There is a body of design and implementation techniques to do this.

Some important portability techniques are based on these principles: the software reflects applications, not implementations; the software assumes machine functions, not features; the design uses logical properties, not physical ones; and the components know only what they need to know. One overall philosophy is that of the *abstract machine.* Some examples of the philosophy are: define data types in terms of the application, define data structures appropriate to the information, distinguish conceptual values from representations, and decouple functional components.

In planning for portability, a good working principle is to expect machines to change faster than applications. Features of a product that are essential to its function can be presumed stable (for example, a data base will always have an enquiry function); features of a product that reflect a specific machine can be presumed unstable (for example, a data base will not always use 2 kilobyte blocks).

[The lecture also discusses an extended example of the design for a portable object base manager.]

23. Technology Evolution: Practice

Reading: none

Two types of change are the introduction of new machines and operating systems, and changes in interface technology. These are examined through two examples.

The first example is the development of a portable IO library that is part of a systems implementation language (BCPL), embodies a philosophy of simple input and output, has evolved through several versions, and illustrates how portability evolves. This history had some influence on other languages, such as C and Ada. The basic concepts discussed are how files are named, how data are transferred to and from files, and special treatment of online user interactions.

Originally, files were identified by *channel numbers*, as in `findinput(6)`. The actual files were specified by the job-control language, which is useful for batch running but not for online running. So later, files were identified by strings, as in `findinput ("SOURCE.DAT")`. This names the file according to the host operating system conventions. However, online programs frequently need to inquire which file to open, so the following convention evolved: `findinput("?Give name of input file ")`. The string is a prompt; the user reply is the file name. On one system, mode information was included in the string: `findoutput ("RESULTS.DAT/PR")`, where the suffix `/PR` meant print after closing. Ada split name and mode into two string parameters.

The basic design views a file as a *stream* of characters, with individual characters are all treated identically. For formatting, some characters were given special meanings, such as `*N` for new line and `*P` for new page. The language implementation created an abstract terminal and printer, with the return key always causing the program to read `*N` and `*N` on output always causing “carriage return line feed.” On record-oriented file systems, each line became a record: output routines interpreted `*N` as end of record and input routines generated `*N` after end of record. On systems with a maximum record size, an escape was sometimes used to indicate that the line is continued in the next record, and to suppress automatic generation of `*N`. The effect was to build a stream abstraction on an underlying record-based IO system.

Originally, the user terminal was unbuffered, so a key pressed was available to the program immediately and a character written to the terminal appeared immediately. Later, the terminal was line buffered; the input was buffered until a carriage return and output was buffered until `*N`. This causes a problem with user dialogue such as `>Give number of turtles:    4`. Language systems have provided several solutions, such as disallowing dialogue with the question and answer on same line, a routine call to force pending output: `forceout()`, or a special character to force output: `...turtles : *F`. The solution usually adopted was called an *interlock*, with reading from the keyboard automatically forcing screen output. This preserved the abstraction of synchronous IO.

The evolution of this IO system shows these features: conservatism (make small changes for preference; add new features within existing framework), abstraction (preserve higher-level concepts and make lower levels conform to them), and implementation hiding (do not burden the user with details of the behavior of the lower levels).

Change in interface technology also provides an example. Recently, there has been a paradigm shift in our view of the user interface from a *command-based* view to a *model-based* view. The paradigm shift had a technological cause, the development of large, fast, bitmapped displays; and it had also an intellectual cause, the exploration of *icons* as representations of entities. The change in technology made possible cost-effective prototypes embodying the new philosophy. The consequential evolution of the software involves new ways the user inputs commands, new ways the system displays responses, and different techniques for designing user interactions.

The older method is based on commands and parameters: the user types a given command, which is followed by parameters naming the objects on which the command is to operate. For example, to delete a file, one might type `DELETE RESULTS.DAT`. The newer method has the user model the operation: point at the object to be affected and perform an operation simulating the required effect. For example, to delete a file, one might move the cursor to the file icon, click to highlight the file, type `%D` to invoke the delete operation, or alternatively drag the file icon to the trash icon.

The principles illustrated include object-oriented (the object affected is indicated first, then the operation to be performed), iconic (objects are not named, they are pointed at), window-based (the various objects are represented by icons distributed in a visible abstract space), and mimetic (the user does not give an imperative command, but rather performs an action for the machine to mimic).

A command-based system usually gives a written response, such as `OK` or `Error 22: you do not have delete access to this file`. The response is part of a dialogue between user and machine. This dialogue is composed of discrete command/response transactions in linear sequence. The user inputs the whole command, and awaits the complete response when the command is done. In a model-based system, the response is different. The relevant system state is visible to the user, and this visible state changes to reflect the new situation; for example, the file icon disappears. Moreover, the state changes continuously as the command is input; the file icon moves along with the cursor to the trash can. Errors are handled by a separate mechanism, such as audible signals.

Command dialogue is often highly modal, in that a major command sets a system mode and each mode recognizes several minor commands. There is a tree structure of commands and subcommands, and commands are explained by a separate facility. Model-based systems should be modeless, in that all commands are always available and a command is always invoked in the same way. Any temporary state should be visible, such as when selected objects are highlighted. Complex commands are issued via menus, and a help facility is integrated with command input.

There are several implementation issues or principles. The software must build or use specific abstractions; for example, each object has a name, icon, command set, and response set, and the object space holds icons in geometrical relationships. The command interface must be richer to understand objects and operations, support abstract operations with diverse implementations, and provide continuous feedback during command entry. The system must maintain a visible model of the processing being done by any program. There are some new quality and performance issues that the implementation must address: all important states must be visible (a highlighted object must never be hidden), the visible model must change smoothly and synchronously (cursor cannot lag behind mouse), and the user must be reassured that the program is still running (long commands must show wait icon or "% done" bar). The last is the hardest to retrofit to an existing product.

Various techniques are used to cope with these issues: object specification tools and techniques, menu and command design tools, and a window manager separate from programs. In general, these lead to a clear separation of abstract objects and operations used by program from the visible icons and manipulations displayed to a user.

[The lecture also introduces a student exercise, the adaptation of a text editor user interface.]

24. Building Long-Lived Software

Reading: Lehman85, Clapp81, Parnas79

Ideally, a software product should remain deployed as long as it is useful, adapt to changing requirements, adapt to changing technology, and remain error free and efficient. This requires continuous corrective and adaptive maintenance. Unfortunately, software often does not endure that long. The reason is that, while it remains useful, it becomes uneconomic, the cost of adaptation increases, maintenance costs increase, and the defect rate rises and efficiency falls. Eventually, the software becomes unmaintainable.

For a typical product, maintenance costs show clear trends: they are initially high as defects are shaken out, declining to a plateau as the product becomes mature, then gradually rising again as the product evolves and loses structure, and eventually reaching the point where cost exceeds benefit. It is this gradual rise in cost that must be delayed or halted. The root cause of this cost increase is degradation of product quality and structure, which can be reduced by planning for longer life, adaptive maintenance techniques, and perfective maintenance techniques.

There must be a good process supporting maintenance, including defect tracking, component stability tracking, and cost tracking. Resources must be budgeted for component replacement, system restructuring, and port analysis. Products should undergo regular perfective maintenance.

To plan for longer life, the planning and design stages are the most important. We must plan for evolution of requirements, plan for changes in technology, plan for regular product revision to improve quality, and design the product to be readily adaptable. There are specific design techniques that help. Most of these have been discussed before: design against a reasonable superset of requirement, specify and document limits of adaptability, hide details of specific or changeable technology, build abstract machines that perform logical functions, and have components know only what they need to know.

For example, consider a product that was designed to sell in several countries and needed to be multilingual. The design called for two special modules, a command decoder and an error message writer. Only these modules knew which language was in use. Each configuration linked the correct pair of modules. As a bad example, consider another product that was developed in one country and later sold in another. Its error reporting code looked rather like this:

```
     IF file_not_found THEN
#ifdef English
     Write('I cannot find ', filename);
#endif
#ifdef German
     Write('Ich kann ', filename, ' nicht finden');
#endif
     END IF;
```

The previous example illustrates two errors: first, the *original* design was at fault in embedding literal text throughout the code, because one can expect error messages to change over time in response to user feedback. Secondly, the *adaptation* made a disastrous mistake; the code should first have been restructured to isolate the error messages.

Adaptive maintenance tends to make products more complex, with an increased number of options or parameters, more complicated case analyses and logic, additional features, and wider and more complex interfaces. However, there are techniques that fight this trend.

For example, consider a product that was ported from a PDP-11 to a PE3200. After the first integration, a number of errors were found with the same root cause: some components assumed they knew the order of bytes in a machine word. Naturally, the byte ordering was different on the two machines. Two obvious solutions are to write alternative components for the new machine (each alternative appropriate to one byte order), and to parameterize the component with a byte ordering flag (the value may be compiled in or set at runtime). Both these solutions are initially cheap, but they increase the product's complexity and hence all subsequent maintenance costs. The solution actually implemented was to rewrite the affected components so that they no longer have to assume byte order. It involved three components and one interface. This was initially more costly, but it simplified all subsequent maintenance and every later port. (This product has been ported to four more machines.)

Consider also an example of a host/target compiler. The code was compiled on the host and downloaded to several targets. The downloaded binary code was sometimes corrupted, and the root cause was that the data link treated some bit patterns (such as #XFFFF and #XFF00) as special and would not transmit them. One solution was to *sanitize* the binary data: look for any bit patterns that might cause trouble, replace them with safe alternatives, and flag the replacements with a special prefix. The downloader program could then modify the data before transmission and reconstruct it after reception. Again, this seemed a cheap solution but at a cost of complicating subsequent maintenance and complicating adaptation to a new communication link. The solution actually implemented was to convert to ASCII—the binary data was encoded into printable characters, with the result that 16 bits became three characters and the amount of data transmitted increased 50%. This required two new modules—encoder and decoder. However, it was robust against almost any future change in the communications link, since almost any link can cope with printable ASCII characters.

One of the principles illustrated by these examples is that where possible, you should choose a solution that simplifies the structure of code, subsequent maintenance and testing, and subsequent adaptations or ports. One approach is to reduce the assumptions made by a component and the special knowledge it requires to operate.

Perfective maintenance is *proactive* change; it is initiated specifically to improve a product. In practice, product quality can be maintained only by regular perfective maintenance. Otherwise, the product degrades and adaptive maintenance increases at the expense of perfective. Several techniques reverse degradation. For example, consider a product that had a large set of user commands. The command set was gradually increasing over time, and the cost of changing or adding a command gradually rose. The root cause was that the command interpreter top level was a CASE statement, with commands considered simple implemented inline, but commands considered complex invoked by subroutines. Accordingly, any scenario followed a complex path. The solution ultimately implemented was to restructure the top levels: all commands invoked subroutines, every arm of the CASE statement had the same pattern, system state was grouped into packages, each subroutine imported only the packages it needed, and transient information was passed as parameters. The result was a substantial decrease in maintenance costs. (The process also uncovered a dozen or so bugs.)

As another example, consider an IO library that was written in a high-level language, but using the same data structures as the underlying operating system (file control block, record descriptor block, device characteristics record). The product was to be ported to another operating system. Virtually every line in the IO library was affected, and the attempt to port was abandoned after three months. The solution ultimately implemented was to split the IO library into two components: a lower level, implementing an abstract IO machine, written in assembler code and interfacing to the operating system; and a higher level, using the abstract machine, written in the high-level language and independent of the operating system. The interface design took one week, and the three new components were written by two people in six weeks. The operating system dependencies had been isolated to 25% of the code.

Consider also the example of a program that was interactive and menu driven, in which the menus were frequently enhanced. For ease of maintenance, they were stored in an editable ASCII text file. However, interpreting this file at run time was costly, and as the menu file grew, the program became more and more sluggish. The solution tried was to define a binary, indexed menu file format; at startup, read the menu text file and build the binary file; and then run the user dialogue from the binary file. This greatly improved program response time, but program startup time was 2 to 3 minutes. The users did not regard this as an improvement. The answer was to split the program into two programs: the *builder*, which creates the binary file and is run in batch whenever menus are changed; and the *driver*, which uses the binary file for the dialogue and is the program invoked by the end user. This combines fast response with rapid startup. Moreover, testing a revision built the binary file “for free”.

The first and key principle illustrated by these examples is to keep track of maintenance costs and their causes. Other points are to allow time for proactive revision of products, restructure to simplify and regroup components, and follow up any major change with a review.

25. Software Quality Assessment

Reading: Collofello87, Green76, Tsichritzis77

Software when deployed should be of good quality, and maintenance must not reduce its overall quality: it must preserve good quality and actively improve poor quality. To do this, we need to know what is software quality, how can it be measured, and how can it be preserved and improved.

Good software has these features: it conforms to all requirements and constraints, it has been created using proper documented standards, and it possesses all implicit qualities of professional software. The last is rather a circular definition! In practice, quality must be measured, and this requires software quality metrics. They must be applicable to products and components.

One can construct a list of features that evidence quality in terms of current utility (efficiency, reliability, and usefulness) and future utility (testability, maintainability, modifiability, portability, and reusability). Similarly, one can list the *factors* that control the features: documentation, component defect rate, component stability, currency against requirement, currency against technology, structural integrity, and complexity. The main underlying factors that affect overall software quality are documentation quality, product complexity, component functional independence, and component environment independence. All of these can be measured and hence assessed.

The main *routine* quality improvement tasks are improving quantity and structure of documentation, reducing unwanted complexity at all levels, and removing unnecessary dependencies. There are also some important *specific* actions, recreating missing documentation and performance and efficiency improvement.

The key to understanding a product is its documentation, including *what* (the function the product is to perform), *why* (the requirement it is to meet), *how* (the way it implements its function), *with what* (the platform it needs in order to run), and *where* (the market it is intended to serve). Subsequent maintenance depends on this understanding. Document quality can be assessed by *existence* tests (whether a particular component is explained, the source of an algorithm is referenced, or the format of a command is documented) and by *traceability* tests (whether we can find all users of this interface, find all modules used by this command, or list all defects found in this component). Document quality can be improved and maintained by recreating missing pieces (reverse engineering), revising data capture procedures, adding relations to the information base, determining the provenance of “orphan” data, and replacing untraceable components. A good environment enforces most of this.

There are many kinds of software complexity: overall structural complexity, overall behavioral complexity, interface complexity, module static complexity, and module dynamic complexity. Each can be assessed and reduced. Structural complexity is measured by the number of relations between components, the depth of the import or dependency graph, the width of the dependency graph, and the existence of tangled cross-dependencies. It may be reduced by regrouping definition modules, hiding lower levels of abstraction better, and revising imperfect abstractions. Behavioral complexity is measured by the number of modules used by one transaction, the amount of each module executed by one transaction, the number of data representation changes, and the number of error messages elicited by one transaction. It can be reduced by shortening the transaction thread length, regrouping transaction processing code, redesigning internal data formats, and rethinking pre- and postconditions. Interface complexity is measured by the number of parameters to procedures, the number of different parameter types, the use pattern of parameter values, and the number of exceptions that can be raised. It can be reduced by using a common form for analogous operations, using common definitions for analogous attributes, and removing or defaulting little-used options. Module static complexity is the infamous “spaghetti code,” the number of control structures and average size, the number of branch and merge points, the number of loop exit points, and the number of function return points. It can be reduced by judicious use of state variables, unnesting of control structures and repetition of code, use of auxiliary functions, and redesign of persistently bad modules. Module dynamic complexity is shown by a lot of persistent internal state, functions with “pernicious” side effects, the need to call several routines in a specific order, and the need to perform initialization and finalization. It can be reduced by exporting visible state objects, using weak preconditions where possible, defining types with automatic initialization and finalization, and replacing error returns with exceptions.

Functional independence is a major factor in the adaptability of software. Components with different functions should interact only when necessary. Changes in one component should not affect others, and changes in shared objects should be upward-compatible. Function interaction can be measured by functions that invoke others at the same level, functions that rely on state set by others, and functions that leave state to be dealt with by others. It can be reduced by defining appropriate functional levels, revising pre- and postconditions, and making objects self-organizing. Component interaction can be measured by correlations between internal states of components, components that repeat tests performed by others, and components that detect errors but pass them downstream. Unwanted interactions can be reduced by resetting component state after all transactions, splitting components by transaction type, and using filters to remove and handle erroneous data.

Defects in object abstraction are shown by operations that must see irrelevant attributes, invariant relations that are hard to preserve, and inaccessible internal state with visible effects. They can be corrected only by better abstraction: proper use of analogy to generalize operations and proper specification of pre- and postconditions. This usually requires substantial revision of the object before the code using it can be improved.

Environment independence is a key factor in the portability of software. One can directly measure portability after a port, and one can estimate portability from representation independence, implementation independence, and sensitivity to resource availability. The portability measure of interest to management may be computed as 1 - cost-of-port/cost-of-rewrite. This is governed by the number of interfaces that change, the number of modules that change, and the size and complexity of changed modules. It can be improved by reducing these numbers (or by creating reusable templates for specific modules).

Representation independence is achieved if operations on an object do not depend on representation. Examples of representation dependencies include the length of a machine word, byte ordering, and order of components in a record. They may be reduced by defining types to reflect application constraints and by accessing data using object-oriented abstractions. Implementation independence is achieved if the effect of an operation does not depend on the algorithm. Examples of implementation dependencies include stable versus unstable sorting, buffered versus unbuffered IO, and rounded versus unrounded floating-point operations. Implementation dependence may be reduced by using weakest preconditions and invariants, tightening of functional requirements, and exact specification of abstract machines. Examples of resource sensitivity include buffers or tables with hardwired maxima, arbitrary limits on the size or quantity of an object, and assumptions that transactions will (or will not) execute in parallel. Sensitivity may be reduced by allowing dynamic growth of all objects or data stores (subject to some overall limit) and by using explicit synchronization where necessary.

26. Reverse Engineering

Reading: Britcher86

Reverse engineering is, essentially, the recreation of upstream objects from downstream ones. The object at hand had a history that is now partly missing. We need to create a plausible reconstruction. The motivation is straightforward: it is necessary to maintain this product.

Maintenance requires a comprehensive product description and history, and if we do not have the history, we must fabricate it. This is all part of the cost of taking on the product. The overall goal is to *integrate* this product into our development and maintenance system. We must create documents appropriate to our own system, complete and with full traceability, that, if part of true history, would have led to what we see. For example, if a company has been given a product to maintain and there is no accompanying design document, it is necessary to reconstruct the design. The goal should be to recreate a design object that comes as close as possible in content to the original but with a style and notation that reflects our practices, not those of the author.

There are other things we may wish to recreate, such as source code from object code, building plan from product structure, requirements from behavior, test sets from behavior, or change requests or bug reports from change logs. However, in most cases, we are starting with source code. Such reverse engineering has severe limitations: rejected designs cannot be recreated, most design rationale is lost forever, and errors in the product will generate erroneous design. In addition, our design notation may be inappropriate—there is no way it could ever generate the code we see.

There are several useful principles, among them analysis, transformation, archaeology, and historical investigation. Code analysis is the process of looking at code to determine the purpose of each component and the reason for its structure. From that analysis, one obtains basic functions, underlying concepts, and a building plan. Code transformation is the conversion of a code object into another object; for example, a design object, "black box" test set, or dependency graph fragment. Many of these conversions can be partly automated. This gives us a missing object and a traceability arc. Software archaeology involves looking at successive *versions* of the code to determine changes over time and to reconstruct reasons for changes. It can also help us understand mature code, because earlier versions may be simpler, earlier versions better reflect original design, earlier functionality is probably more important, and unstable fragments can reveal design problems. Historical investigation tries to reconstruct history from external sources, such as interviews with the authors, interviews with the customers or users, published articles, related products, and minutes of reviews and discussions. The pieces of information are fitted together into a picture of the missing object.

Before beginning the maintenance task, we should determine what comes with the product and find out what is needed but missing. Then we can determine the feasibility of reverse engineering, estimate the cost of reverse engineering, estimate the quality of existing and recreated objects, and decide whether some restructuring or rewriting is needed. It is not always easy to estimate these costs. The cost can be significantly reduced by tool support, such as browsers and annotators, version differencers, transformation tools (especially code into design), and test set generators. However, a lot of human detective work is necessary also.

27. Software Performance Improvement

Reading: none

The objective of performance maintenance is to maintain the efficiency of the product. Efficiency is not just absolute performance; it is performance relative to application and technology. The customer expects performance to remain constant over time for constant technology and to improve as technology evolves. Maintenance must address these expectations.

These are ways of measuring performance, including the size of the information base supported, the number of transactions per second, the time to perform a transaction, and the latency before system response to an event. The appropriate metrics are the ones the *customer* cares about. The required values depend on application context. Also, the average case or the worst case might matter more.

These are the system features that govern performance: IO bandwidth, the rapidity of access to the information base, memory size, processing power, and interrupt or task latency. It is necessary to determine which drivers are important. Some applications are compute bound, so processor speed is the sole performance driver; other applications are IO bound, so IO bandwidth is the driver. Usually, the issue is not so clear cut.

Maintenance should be able to cope with these changes: gradual increase in the size of the information base, gradual rise in the transaction number or rate, gradual increase in the number of simultaneous transactions, stepwise speed and capacity increases in components, and abrupt changes in performance characteristics. This usually requires advance planning and specific responses to major changes. Overall, one might expect that an increase in traffic reduces performance, and an increase in capacity improves performance. This is broadly true, but, unfortunately, the detailed relationship is critical to the behavior of the product. An increase in traffic (transaction rate, message size, etc.) may have various effects on performance: almost no effect, gradual decrease, proportionate decrease, disproportionate decrease, or abrupt collapse at a critical threshold. To avoid a crisis, it is necessary to predict the effect before the product becomes overloaded. Similarly, an increase in system capacity can have little effect or none, proportionate increase in performance, disproportionate increase, or critical increase as a threshold is crossed. Again, one must analyze the effect of changing each major system component.

Software maintenance must first maintain performance of the system as deployed and used. It must allow for gradual growth in traffic, know the effect of growth on key performance drivers, ensure that projected growth stays below critical thresholds, compare actual growth with projected growth, and plan for timely revision before growth limits are met. For example, what growth should a file store system expect? A rapid growth in physical disc size might lead to rapid growth in directory size or abrupt increases in maximum file size. A moderate growth in the number of users may lead to moderate growth in files-per-user and moderate growth in transaction parallelism. Appropriate maintenance actions might be to implement size-insensitive directory algorithms, check against simulated growth, and collect statistics of actual usage; plan for higher parallelism by making most components reentrant, minimizing the time the directory is locked, and requiring multi-accessed files to be specially marked; and anticipate a critical threshold as parallelism grows and plan to revise the product well in advance.

Maintenance must next try to maintain efficiency as capacity changes yield performance changes. This again implies knowing the key performance drivers, recommending upgrades in critical components, maintaining the system in overall balance, and delaying insertion of new technology that is not needed. One should try to anticipate major innovations. For example, consider maintaining a file store as technology evolves: a rapid fall in cost of memory means buffers and caches can get bigger, CPU speed increases faster than IO speed so that IO bound programs will become more so and CPU bound programs will suddenly hit IO limit. The obvious action is to improve IO caching with larger caches, and with better and more adaptive algorithms. Implementation involves user customization of cache size and algorithms, IO system self-monitoring and self-adapting, and usage statistics collected on IO performance. Also, track product performance to anticipate when another product might hit the IO bottleneck.

Maintenance should then try to improve efficiency, yielding better performance with existing capacity. This is simply performance tuning after deployment, and the key principle is the same as before: find the bottleneck, fix it, find the next bottleneck. However, one has the advantage of actual usage data. For example, consider a compiler that reused a general-purpose storage allocator and garbage collector. It was very slow, and profiling showed that the product spent over 40% of its time doing storage allocation and deallocation. This was a clear candidate for perfective maintenance. The actions were based on *actual* usage patterns. The compiler assumed all allocated storage contained garbage; the change was to remove elaborate default initialization of objects. 90% of objects had one of three fixed sizes; the change was to replace a generalized “buddy algorithm” with FIFO free lists.

80% of deallocations occurred at the end of a major phase; the change was to replace garbage collection with block deallocation. The result was a new storage allocator about 20 times as fast and about one-fiftieth the size.

28. Software Perfectability

Reading: none

It is possible to improve code quality after deployment. This requires proactive maintenance by dedicated people. It should focus on the most defective components, the major causes of defects, and the most beneficial remedial actions. The realistic goal is a constant high level of quality with an affordable use of resources. The key rule is always to take the most effective action. We would like to maximize benefit and minimize cost. What are the limits of the achievable?

One major goal of perfective maintenance is to reduce subsequent maintenance costs. Rather than correct defects, reduce the *propensity* for defects; rather than continually adapt code, increase its *generality*; rather than create new configurations, improve *portability*. Good perfective maintenance involves deciding what attribute to change, selecting a metric to measure change, collecting appropriate data, selecting the target for change, implementing the change, and tracking the result of the change.

One should change attributes that affect cost or benefit, such as defect propensity, portability, adaptability, and scalability. The attribute chosen depends on product and marketing plans.

Before trying to change something, quantify it: determine what the unit of measurement is, how to take a measurement, what its present value is, what value we seek to achieve, and what the equation is relating cost or benefit to this value. Data must be collected, or measurements made according to the metric for the product in question in sufficient detail to isolate targets for change. In practice, this means that each datum should be tied to a specific component, a specific part of the life cycle, and a specific part of the requirement.

Target selection is the critical part of the process: deciding what to change to effect the improvement. There are several guidelines that help, including Pareto analysis, cause/effect analysis, and leverage analysis. Pareto analysis is based on the principle that a few of the X exhibit most of the Y; for example, “20% of the modules contain 80% of the bugs.” If it costs the same to rewrite any module, then we should rewrite that 20%. The analysis assigns defects to components and selects those with the highest number of defects. Cause/effect analysis determines the root cause of each instance. It relies on the theory that some few causes are responsible for most of the effects; for example, “Half the bugs are caused by uninitialized variables.” The most prolific causes are identified and dealt with (note that in the quoted example the bugs will be scattered through many components). Leverage analysis is based on the principle that small changes can have large consequences. For each defect, a possible corrective change is identified. Because some changes will correct more than one defect, the change to perform first is the one that corrects most.

Change will usually involve some rewriting or restructuring of the product. It should be performed as part of the normal version development process, and the revised product is reissued as a new version. Data should be collected for the new version to demonstrate that the change has worked.

Unfortunately, there are other forces that set limits to software perfectibility, such as diminishing returns, benefit tradeoffs, and opportunity costs. Diminishing returns is the downside of the Pareto principle. For example, revise 20 modules and fix 80 bugs, revise 16 more modules and fix 16 more bugs, revise 13 more modules and fix 3 bugs. At some point, cost will exceed benefit. But if several techniques are applied, the level where further improvement is unreasonable is very low. Benefit tradeoffs are required because some qualities can be improved only at the expense of others. Consider the tradeoff between portability and efficiency. Machine code can be 100% efficient and 0% portable. Typically <10% machine code can yield >80% efficiency. Fully portable code might be very inefficient. Unfortunately, this tradeoff is a hard one, because the benefits accrue to different parties. In practice, maintenance resources are scarce and every use of them therefore has an *opportunity cost*—the cost of not doing the other things that need doing. Resources are deployed on one product only until it becomes more profitable to redeploy them. This will happen even though there is still beneficial work to do.

What is achievable at reasonable cost? Consider two ideal situations: zero defect rate (code that is completely bug free) and total portability (code that can move without change to a new machine).

Here are some historical data, in defects per million lines of code:

Defects/MLOC	Development method
60,000	code developed normally and untested
10,000	code developed normally and tested
1,000	the above plus code and design inspections
300	the above plus formal design and verification
100	the above plus “defect cause removal”
40	software “cleanroom” experiment
4	claimed limit of feasibility

Unfortunately, maintenance is too late for some of this. Some recommended techniques we can adopt are comprehensive acceptance testing, regression testing, operational simulations, inspection of all revisions, some defect cause removal, introduction of formalism in data specifications, and statistical quality control. This should improve about tenfold both defect detection and the rate of introduction of new defects.

Here are some estimates from the history of a compiler for costs in man-days and efficiency of product.

Efficiency	Cost	Development or maintenance effort
100%	400	original compiler
100%	120	new code generator for new machine
70%	60	adapting code generator from template
10%	20	writing machine-code interpreter
3%	4	porting interpreter written in existing language

Realistically, code written in standard languages can be 100% portable, so this is the best route to good portability. However, this assumes a target environment that has already been built up to a certain level of abstraction. Other products must build their own abstractions. A pessimistic estimate is that each layer of abstraction supports about five times its cost in portable code. Few products can afford the efficiency cost of many layers.

29. Final Review (Part 1)

[This lecture reviews software creation (planning, cost, and schedule; and techniques), software testing (correctness, performance, quality), and software deployment.]

30. Final review (Part 2)

[This lecture reviews software maintenance (procedures, types of maintenance, and techniques), and building long-lived software (technical issues and quality issues).]

31. Final Examination

Bibliography

Arthur88 Arthur, L. J. *Software Evolution*. New York: John Wiley & Sons, 1988. Required textbook for course.

Bastani85 Bastani, Farokh B. “On the Uncertainty in the Correctness of Computer Programs.” *IEEE Trans. Software Engineering SE-11*, 9 (Sept. 1985), 857-864.

Beizer84 Beizer, Boris. *Software System Testing and Quality Assurance*. New York: Van Nostrand Reinhold, 1984.

Boehm81 Boehm, Barry W. *Software Engineering Economics*. Englewood Cliffs, N. J.: Prentice-Hall, 1981.

Boehm88 Boehm, Barry W. “A Spiral Model of Software Development and Enhancement.” *Computer 21*, 5 (May 1988), 61-72.

Britcher86 Britcher, Robert N. and Craig, James J. “Using Modern Design Practices to Upgrade Aging Software Systems.” *IEEE Software 3*, 3 (May 1986), 16-24.

Clapp81 Clapp, Judith A. “Designing Software for Maintainability.” *Computer Design 20* (Sept. 1981), 197-204.

Collofello87 Collofello, James S. and Buck, Jeffrey J. “Software Quality Assurance for Maintenance.” *IEEE Software 4*, 9 (Sept. 1987), 46-51.

Glass81 Glass, Robert and Noiseux, R. A. *Software Maintenance Guidebook*. Englewood Cliffs, N. J.: Prentice-Hall, 1981.

Grady87 Grady, Robert B. “Measuring and Managing Software Maintenance.” *IEEE Software 4*, 9 (Sept. 1987), 35-45.

Graham77 Graham, R. M. “Performance Prediction.” *Software Engineering, An Advanced Course*, Bauer, F. L., ed. New York: Springer-Verlag, 1977, 396-463.

Green76	Green, T. F., Schneidewind, N. F., Howard, G. T., and Pariseau, R. J. "Program Structures, Complexity and Error Characteristics." *Proc. Symp. Computer Software Engineering*. New York: Polytechnic Press, Apr. 1976, 139-154.
Holbrook87	Holbrook, H. B. and Thebaut, S. M. *A Survey of Software Maintenance Tools That Enhance Program Understanding*. Tech. Rep. SERC-TR-9-F, Software Engineering Research Center, University of Florida, Gainesville, Fla., Sept. 1987.
Ince85	Ince, D. C. "A Program Design Language Based Software Maintenance Tool." *Software—Practice and Experience 15*, 6 (June 1985), 83-94.
Kernighan78	Kernighan, Brian and Plauger, P. J. *The Elements of Programming Style*. New York: McGraw-Hill, 1978.
Lehman85	Lehman, Manny M. *Program Evolution: Processes of Software Change*. Academic Press, 1985.
Martin83	Martin, James and McClure, Carma. *Software Maintenance: The Problem and its Solutions*. Englewood Cliffs, N. J.: Prentice-Hall, 1983. Required textbook for course.
Martin85	Martin, James and Leben, J. *Fourth Generation Languages*. Englewood Cliffs, N. J.: Prentice-Hall, 1985. Published as two volumes; second volume published in 1986.
McCabe76	McCabe, Thomas J. "A Complexity Measure." *IEEE Trans. Software Engineering SE-2*, 4 (Dec. 1976), 308-320.
Myers79	Myers, Glenford J. *The Art of Software Testing*. New York: John Wiley & Sons, 1979.
Ng90	*Modern Software Engineering; Foundations and Current Perspectives*. Ng, P. A. and Yeh, Raymond T., eds. New York: Van Nostrand Reinhold, 1990.
Parnas79	Parnas, David L. "Designing Software for Ease of Extension and Contraction." *IEEE Trans. Software Engineering SE-5*, 2 (Mar. 1979), 128-137.
Poole77	Poole, P. C. and Waite, W. M. "Portability and Adaptability." *Software Engineering, An Advanced Course*, Bauer, F. L., ed. New York: Springer-Verlag, 1977, 183-277.
Prell84	Prell, Edward M. and Sheng, Alan P. "Building Quality and Productivity into a Large Software System." *IEEE Software 1*, 3 (July 1984), 47-54.
Pressman87	Pressman, Roger S. *Software Engineering: A Practitioner's Approach*, 2nd Ed. New York: McGraw-Hill, 1987. Required textbook for course.

Sha89	Sha, Lui and Goodenough, John B. *Real-Time Scheduling Theory and Ada*. Tech. Rep. CMU/SEI-89-TR-14, ADA211397, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Apr. 1989.
Tsichritzis77	Tsichritzis, D. “Reliability.” *Software Engineering, An Advanced Course*, Bauer, F. L., ed. New York: Springer-Verlag, 1977, 319-373.
Turner86	Turner, David. “An Overview of Miranda.” *SIGPLAN Notices 21*, 12 (Dec. 1986), 158-166.
Wegner84	Wegner, Peter. “Capital-Intensive Software Technology.” *IEEE Software 1*, 3 (July 1984), 7-45.

3.5. Software Verification and Validation

Students' Prerequisites

Students should have familiarity with programming-in-the-small in block-structured languages (such as Ada, C, or Pascal). They should be able to read and write first order predicate calculus formulae and know how to prove simple theorems, such as DeMorgan's Laws.

Objectives

Students will learn how to evaluate software for correctness, efficiency, performance, and reliability. Specific skills they will master include program proving, code inspection, unit-level testing, and system-level analysis. In addition, students will gain an appreciation for the difficulty and cost of some types of analysis and the need for automation of tedious tasks.

Philosophy of the Course

This course stresses problem-solving skills, especially in analysis of code. Students practice several different kinds of analysis, including proofs, testing, and inspections. One main example is used throughout the course so that students can see the difference in methods applied to the same subject. Part of the course deals with attitudes in industry toward reliability and performance.

Syllabus

The syllabus assumes 30 class meetings, including midterm and final examinations. Each meeting is planned to include approximately 55 to 60 minutes of lecture and 20 minutes of class discussion.

1. Introduction
2. Reviews: Walkthroughs
3. Reviews: In-Class Review
4. Reviews: Inspections
5. Reviews: In-Class Inspection
6. Reviews: Cleanroom
7. Proofs: Natural Deduction
8. Proofs: Invariant Assertions
9. Proofs: Arrays
10. Proofs: Procedures
11. Proofs: Weakest Preconditions
12. Proofs: Symbolic Evaluation
13. Proofs: Program Development
14. Midterm Review
15. Midterm Examination
16. Unit Testing: Overview
17. Unit Testing: Structural
18. Unit Testing: Dataflow Analysis
19. Unit Testing: Partition Analysis
20. Unit Testing: Coverage
21. Unit Testing: Limitations
22. System Testing: Integration and System Testing
23. System Testing: Performance Analysis
24. System Testing: Regression
25. System Testing: Performance Testing
26. System Testing: Environments and Tools
27. System Testing: Safety
28. System Testing: Metrics and Reliability
29. Final Review
30. Final Examination

Summaries of Lectures

1. Introduction

Reading: Beizer84, Chapters 1-2

Quality of a software product manifests itself in many dimensions. In the operations dimension, we can consider correctness (does it do what I want?), reliability (does it do it accurately all of the time?), efficiency (will it run on my hardware as well as it can?), integrity (is it secure?), and usability (can I run it?). In the transition dimension, we can consider portability (will I be able to use it on another machine?), reusability (will I be able to reuse some of the software?), interoperability (will I be able to interface it with another system?). In the revision dimension, we can consider maintainability (can I fix it?), flexibility (can I change it?), and testability (can I test it?).

Quality exists throughout a product, not just in the source code. This includes the requirements specification, the functional specification, the design, the implementation, the test plan, and the documentation.

Validation is the process of determining that a product fulfills its requirements. (Is it the right product?) Verification is the process of determining that a product agrees with its specification. (Was it done right?) Verification can be formal (based on mathematics, including logic; validity of statements can be mechanically checked) or rigorous (strictly following rules; compliance can be audited).

2. Reviews: Walkthroughs

Reading: Weinberg84

Software technical reviews come in many forms and should be conducted for several reasons. Proofs of correctness are too hard, and testing is not sufficient. Some objects, such as requirements, cannot be proved correct or tested. Reviews provide feedback about the development process, and they are educational.

The most common review types or methods include: walkthroughs, which focus on objects; inspections, which focus on aspects; audits, which are a solitary exercise focusing on defects; round-robins, which cycle through participants; and informal. The objects reviewed include requirements (to assess comprehension and completeness), specifications (to assess consistency and completeness), designs (to assess integrity), code (to assess accuracy), documentation (to assess accuracy and clarity), and other work products.

A review is formal if its results are reported in written form, it is a public event, and all participants are responsible for the quality of the review. The advantages of formal reviews include that its reports serve as milestones for the project, the publicity encourages preparation, and personality conflicts can be addressed openly. The disadvantages are the perception that too much reviewing leaves little time for “real work,” and that formal meetings are sometimes too impersonal.

The preparation for conducting a walkthrough begins with selection of the participants. These should include a review leader (or coordinator or moderator), a recorder (or secretary or scribe), a presenter (or producer), other producers, and possibly user representatives, external reviewers, and others. Then the walkthrough is scheduled, an agenda prepared, and the materials are studied.

The review leader is responsible for scheduling the review, conducting the review (especially maintaining order, keeping the focus on the review, and assuring that all items are covered), preparing the report, and following up on any action items. The recorder is responsible for recording items of interest (especially defects and anomalies discovered and any action items discovered), and assisting the leader in preparing the report. The presenter is responsible for preparing carefully for the review, being objective, and being accountable.

The report of the review should include a management summary, typically one page that describes what was reviewed, who participated (with signatures), and results of the review (as acceptance or rejection of the product). The main part of the report addresses technical issues: it lists all defects and anomalies and records action items. A related issues section may be used as a minority report; to remove contentious items from the technical issues section; and to address problems not solvable by action items.

3. Reviews: In-Class Review

[This class period is used to conduct a review, rather than a lecture. The first part of the next class period is used as a debriefing to discuss what happened in the review, why, and what should have happened.]

4. Reviews: Inspections

Reading: Fagan76

A well-managed software process has identifiable stages. Exit criteria are used to recognize passage between stages; inspections are used to ensure satisfaction of exit criteria. Walkthroughs and inspections differ in that a walkthrough is a developer's method, the focus is on the object, and the objective is understanding; whereas an inspection is a manager's method, the focus is on aspects of the product, and the objective is detection of defects. Audits and inspections differ in that inspections are conducted early in the process, they collect data to measure the process, and the objective is defect detection and removal; whereas audits are conducted late in the process, they use data from the inspections, and the objective is monitoring.

The inspection process includes planning, presenting an overview of the product, preparation, the inspection itself, rework, and follow-up. The overview may be conducted by the producer, and its object is education. One overview may be sufficient for several subsequent inspections. The preparation includes studying distributed materials, studying previous inspections, studying checklists for finding defects, identifying some defects and anomalies, and noting these or other troublesome areas in a preparation log. The inspection consists of an introduction, reading the material, recording defects, reviewing the defect list, and completing the reports. Rework removes defects or resolves them in some other way. The follow-up assures that all defects and issues have been resolved.

The roles of the participants are similar to those for a walkthrough. The review leader schedules the review, conducts it (maintains order, makes rulings on defect classification, keeps the focus on the review, and assures that all items are covered), prepares reports, and follows up on defects and action items. The recorder records all items of interest, including defects and anomalies discovered and action items generated, and assists in the preparation of reports. The reader presents materials during the review, usually a paraphrasing of low-level detail. The reviewers must prepare carefully for the review, be objective, and be accountable. Managers do not attend inspections, so the discussion can be candid.

The report consists of the preparation log (which lists potential defects, troublesome areas, and other information such as the amount of time spent), a management summary (usually one page describing what was reviewed, a list of participants, and a description of the results of the review as acceptance or rejection of the product), a defect list (including a classification of each), a summary of defects (totals by type; useful to management to measure the processes of development and inspection).

5. Reviews: In-Class Inspection

[This class period is used to conduct an inspection, rather than a lecture. The first part of the next class period is used as a debriefing to discuss what happened in the review, why, and what should have happened.]

6. Reviews: Cleanroom

Reading: Dyer87, Selby87; Linger88 and Mills86 for background

The cleanroom approach to software quality has its foundations in structured programming (a restricted set of primitive control structures and stepwise refinement), functional correctness, and statistical sampling theory. The method uses incremental development, an approach derived from stepwise refinement; it requires that specifications be structured; it insures that the unit of verification is relatively small; and it allows for overlap of development and testing. Formal methods are used in specification and design. Functional verification is used instead of debugging: each language element has a verification rule (functional description), verification is composed of function compositions, and it is goal-directed and mostly bounded. After coding, statistically-based independent testing is used rather than unit testing by the developers. The method is based on sampling of users' frequency of operations; additional tests are added to ensure coverage of all functionality; testing is performed by an independent group, a target "mean time to failure" (MTTF) rate is chosen as the criterion for completion of testing.

Comparison of cleanroom with other methods shows that it substitutes formal verification for debugging and unit-level testing; it emphasizes the importance of frequency of users' operations; and the role of testing is to increase confidence by statistical means. A University of Maryland study of 10 cleanroom teams vs. 5 non-cleanroom teams showed that the former delivered more product and made all intermediate deliverables, used a smaller subset of the programming language, and passed more tests. An IBM effort to build a COBOL restructuring tool reported significantly fewer defects in the delivered product. A 1980 census software system of 25,000 lines of code developed with this method "...ran throughout the production of the census (almost a year) with no errors detected."

7. Proofs: Natural Deduction

Reading: Gries81, Chapters 0-4

Some of the basic definitions include *axioms*, a recursive set of valid wffs (well-formed formula); *rules*, a finite set of mappings from valid wffs to valid wffs; and *proof*, a sequence of wffs, in which each wff is an axiom or is derived from wffs by rules. The last wff is said to be *deduced* or *proven*. A fundamental theorem is that every deducible wff is valid. These definitions may apply to *systems* (collections of wffs). A *model* is an interpretation that makes all wffs of a system true. That a wff is valid is equivalent to its negation being unsatisfiable. Theorem provers try to demonstrate validity of an assertion by showing the unsatisfiability of its negation; natural deduction systems try to show the validity of the assertion directly.

Gries has introduced a variation of this approach in which there are no axioms; there are 10 rules (an introduction rule and an elimination rule for each operator); premises may appear as wffs in a proof; and premises and derived wffs may only be used if they are on a previous line in the same scope or on a previous line in an enclosing scope.

There are several important solvability and decidability results. Validity of wffs in propositional calculus is decidable. Validity of wffs in first-order predicate calculus is undecidable, but partially solvable. Validity of wffs in second-order predicate calculus is undecidable. First-order predicate calculus has a complete deduction system, but second-order does not. Validity is decidable if a Turing machine can decide if a wff is valid; validity is partially solvable if a Turing machine will (eventually) output "yes" to every valid wff. A complete deduction system is one that can be used to prove every valid wff.

Another important proof method is mathematical induction.

[The lecture includes a detailed discussion of inference rules and examples from Gries81.]

8. Proofs: Invariant Assertions

Reading: Hoare69

Program verification (and its foundations) is a very old idea; it was addressed by Aristotle and Euclid, by Ada Lovelace in the 1840s, and by Turing in the 1940s. It can be approached in various ways, including invariant assertion (Hoare), weakest precondition (Dijkstra), and functional (Mills). Some basic definitions include *assertion*, a valid wff that is interpreted relative to its location within a program; *precondition*, an assertion before a statement or program component; and *postcondition*, an assertion after a statement or program component. A *program proof* is a program annotated with assertions such that each assertion is justified by one of the following: it is a precondition of the program; it follows from the previous assertion by logical consequence (using first-order predicate calculus); or it follows from the previous program statement by application of a program inference rule. *Partial correctness* is defined to mean that if the program terminates, then it is correct; stated another way, if the precondition is true and the program terminates, then the post-condition will be true. *Total correctness* is defined as partial correctness and termination is proven.

Proof of termination of a loop is based on the idea of defining a function from the product of the data types of variables used in the loop to the natural numbers, and then showing that the function is strictly decreasing and bounded below.

[The lecture includes discussions of many detailed examples of definitions; of inference rules including the rule of composition, rules of consequence, assignment axiom, condition rule, and iteration rule; and examples of verification of multiplication and square root programs.]

9. Proofs: Arrays

Reading: Gries81, Chapters 5-6

Proofs involving programs with arrays require some new and additional rules. Quantifiers for arrays include a universal quantifier with notation (Ai: R: E), meaning for all i satisfying R, expression E is true; an existential quantifier with notation (Ei: R: E), meaning there exists an i satisfying R, such that expression E is true; and a counting quantifier with notation (Ni: R: E), meaning the number of i satisfying R, such that expression E is true. A *section* of an array is a statement of the form X[a:b], meaning (Ai: $a \le i \le b$: X[i]). When $b < a$, X[a:b] is empty; anything can be asserted about the empty section.

[The lecture includes a detailed example of proofs involving arrays, using a sorting program.]

10. Proofs: Procedures

Reading: Gries81, Chapter 12

Proofs involving procedures require the introduction of several new rules, including rules for nonrecursive procedures, a rule for local declarations, and a rule for recursive procedures. These include rules for procedure with no parameters, procedures with actual parameters the same as the formal parameters, the rule of substitution, the rule of declaration, the rule of substitution and restrictions on it, and the rule of adaptation and restrictions on it.

[The lecture includes detailed discussions of the proof rules and their notations. It also includes detailed examples of proofs involving a nonrecursive procedure and a recursive procedure.]

11. Proofs: Weakest Preconditions

Reading: Gries81, Chapters 7-11

The weakest precondition approach to correctness proofs requires the introduction of new notation, including the conditional boolean operators *cand* and *cor*; the *domain*, which is the set of all states where an expression is defined; the empty statement *skip*; multiple assignment notation; case statement notation; and while loops and guards. A new concept is the implementation concept of evaluation. The proof rules for weakest preconditions include assignment, conditional, iteration, composition, and consequence.

[The lecture includes a detailed discussion of an example of a multiplication program.]

12. Proofs: Symbolic Evaluation

Reading: Hantler76, Kemmerer85

Symbolic execution uses a standard mathematical technique of letting symbols represent arbitrary program inputs. What we need is a machine that performs algebraic manipulation on the symbolic expressions. Inputs are represented symbolically. The program *state* includes the values of variables, the program counter, and the path condition P.

A rule is defined for each statement in the language. For example, for the assignment statement X := E, the rule is to replace all variables in the expression E with their current values. The resulting expression becomes the new value of variable X. The program counter is set to the next instruction and path condition is unchanged.

The Unisex tool is a verification condition generator. It supports symbolic evaluation by automating path condition propagation.

For a statement of the form "assume (<boolean expression>)" the rule is that variables in the boolean expression are replaced by their current symbolic values. Let B represent the resulting expression. Then the new path expression is defined by $P_{new} = P_{old}$ & B.

For a statement of the form "prove (<boolean expression>)" the rule is to let B represent the symbolic expression that results from replacing all the variables in the boolean expression with their current values. Then if $P \Rightarrow B$, the expression is verified; otherwise it is not.

It is common to use a tree structure to represent the symbolic execution, where each node represents a statement in the program and each branch point corresponds to a forking IF statement.

For a statement of the form "WHILE <boolean expression> DO <statement>" the rule is to let B represent the evaluated boolean expression. If $P \Rightarrow B$ then execute the statement list followed by the same WHILE statement; if $P \Rightarrow \sim B$ then execute the statement following the WHILE construct. If $P \Rightarrow B$ and $P \Rightarrow \sim B$ then two cases must be considered. In the first case, where B is true, $P_{new} = P_{old}$ & B and then execute the statement list followed by the same WHILE statement. In the second case, where B is false, $P_{new} = P_{old}$ & B and then execute the statement following the WHILE construct.

For every statement in the language we must define its effect on the state (the effect on the variables, program counter, and path condition).

13. Proofs: Program Development

Reading: Gries81, Chapters 13-17

[Guest lecturer: David Gries, Cornell University]

Algorithms and programs can be developed simultaneously with proofs of their correctness. The algorithm must first be specified rigorously.

The *loop invariant* is an important concept for developing correct loops in programs. At least an approximation for the loop invariant should be developed before the loop.

Two heuristics for finding the invariant are replacing an expression in the postcondition by a fresh variable and deleting a conjunct of the postcondition.

In the notation $\{ Q \}$ **do** $B \rightarrow S$ **od** $\{ R \}$, Q is the *precondition*, B the *guard*, S the *loop body statement(s)*, and R the *postcondition*. Let P be the loop invariant for such a loop. Find the guard B by solving for it in $P \wedge \neg B \Rightarrow R$. The first step in developing the body S is to determine how to make progress. Given **do** $B \rightarrow$ ___ progress **od**, precede "progress" by a statement S' satisfying $\{ P \wedge B \}\ S'\ \{ wp(\text{progress}, P) \}$. If the body of the loop is too inefficient, try to strengthen the loop invariant so that the body can be improved.

To prove $\{ Q \}$ **do** $B \rightarrow S$ **od** $\{ R \}$ using invariant P and bound function t, prove: (0) $Q \Rightarrow P$; (1) $\{ P \wedge B \}\ S\ \{ P \}$; (2) $P \wedge \neg B \Rightarrow R$; and (3) the loop terminates, meaning that execution of S decreases t and $P \wedge B \Rightarrow t > 0$.

[The lecture includes examples of application of these ideas to the development of some simple algorithms and programs.]

14. Mid-term Review

15. Mid-term Examination

16. Unit Testing: Overview

Reading: Beizer84 chapters 3-4; Beizer83 and Myers79 for background

Software testing is conducted at many phases of the development process. Unit testing is the lowest level, and it is conducted relatively soon after implementation of a module or unit. Integration testing follows successful unit testing; system testing is performed on the entire, integrated system; and acceptance testing is conducted by the user just prior to taking delivery of the system. Regression testing is conducted during maintenance.

Some of the methods of unit testing are classified as black-box or white-box, or as top-down or bottom-up. Top-down testing simulates unwritten pieces with stubs and replaces stubs with real pieces as they become available. Bottom-up testing simulates unwritten pieces with drivers and replaces drivers with real pieces as they become available. Black-box or functional testing is based on the assumption that the inner details of a unit are unknown, and the tests are chosen in terms of specifications of the unit. White-box or structural testing is based on the assumption that the inner details of the unit are known, and the tests are chosen in terms of the structure of the unit.

Some black-box testing methods are equivalence partitioning (divide the input into equivalence classes; each test point in a class tests the same properties of the system, so select only one test point from each class); boundary-value analysis (select test points at boundaries of input data equivalence classes—on either side and right on the boundary, and do the same for output classes); syntax testing (describe the input in terms of a language, create artificial test points from boundary values of grammar); logic-based testing (create decision tables from the specification, and choose test points to ensure coverage of the table); cause-effect graphing (similar to decision tables, but uses a graph instead to show combinations of decisions); and error guessing (try to second-guess the developer of the code and generate test cases for likely errors).

Some white-box testing methods are symbolic execution (assign initial symbolic values to variables and inputs, execute the program by assigning symbolic values instead of evaluating expressions, keep track of possible paths and try each segment); statement coverage or branch testing (execute each statement in the code, identify unreachable code); path testing (execute each path in the code, some paths may be infeasible); transaction flows (invent a higher-level unit of control flow based on the logical transaction of the system, execute each path in the higher-level description); and state-transition testing (construct a state-machine description of the system and execute each path through the transition graph of the state machine).

Test plans are an important part of testing. They should describe the responsibilities and strategies, and they should integrate with the project management plans. A test plan typically includes a section on objectives (what is accomplished by testing) that may be different for each phase; completion criteria (when testing can stop, if errors are still present, what else must be done); schedules (when the testing will be done, and the dependencies on other project activities); responsibilities for designing tests, running tests, evaluating results, and responding to errors found; strategies (testing method, integration strategy); version control; and the tests themselves (descriptions, test inputs, expected results, and actual results).

[The lecture also includes a detailed discussion of the triangle example from Myers79 and a multiplication example.]

17. Unit Testing: Structural

Reading: Howden76

One form of structural testing is based on the concept of a *path*, which is a sequence of statements that correspond to some flow of control. If there is no input that can cause a path to be executed, then the path is *infeasible*. The subset Di of the input domain D that causes a particular path Pi to be executed is called the *path domain* of Pi. The sequence of computations i = (Pi) performed on a path Pi is called the *path computation*. Two computations i and j are *equivalent* (i = j) if i and j are defined over the same subset D' of D, and i(x) = j(x) for all x in D'.

Program correctness can be defined in terms of path computations. Suppose that P* is a correct version of program P, and that there is an isomorphism between the paths of P and the paths of P*, such that $\forall$ i [D(Pi) = D(P*i) $\wedge$ (Pi) = (P*i)]. Then P is correct. Incorrect programs can exhibit several types of errors. A computation error exists if some computation is incorrect. A domain error exists if some input causes execution of the wrong path. A subcase error exists if some path is missing.

P-Testing is a form of structural testing. Suppose that it were possible to identify all of the paths (a potentially infinite set), and that it were possible to find input data to execute each of these paths. Selecting one input value for each path is called *P-testing*. For computation errors, P-testing is reliable only if computations on a path are either correct for all input or incorrect for all input. For domain errors and subcase errors, P-testing is reliable only if the domains of incorrect paths are completely disjoint from the paths of correct programs.

In summary, P-testing is impossible because it is not possible to determine if the program terminates, and the number of tests may be infinite. P-testing is not reliable.

[The lecture also includes detailed examples of a program, its path domains, its path computations, the error types, how errors of each type may be found.]

18. Unit Testing: Dataflow Analysis

Reading: Rapps85

Dataflow analysis is a useful approach to structural testing. Dataflow anomalies may reveal errors. Path testing is impossible because of the potentially infinite number of paths. In enumerating paths, some paths are infeasible, but that cannot always be determined. Dataflow analysis may discover the more "interesting" paths.

A fundamental concept is "def/use" (definition and use). A *basic block* contains no transfer of control statements; it is represented by a single node in a program graph; and definitions and uses within the block (local) are hidden. Def/use concepts for variables include: an input statement "read V" defines V; an assignment statement "V := X" defines V and uses X; an output statement "print V" uses V; a conditional statement "if p(V) then goto m" uses V. Def/use concepts for paths include: a path containing no def's of a variable V is a *def-clear* path with respect to V; the existence of a def-clear path before the first use of a variable may be an error; and two def's of the same variable without an intervening use is probably an error. A *du-path* is a path from a definition to a use.

Several criteria and sets of criteria for testing have been proposed. The simplest include: *all-nodes* (every node is executed; same as *statement coverage*); *all-edges* (every edge is executed; same as *branch testing*); *all-defs* (every definition is executed and later used); *all-uses* (for each definition, for each use of that definition there is a path tested); and *all-du-paths* (for each definition, for each du-path from that definition there is a path tested).

Variables may be used in predicates or computationally. An example of *p-use* is in the conditional statement "if p(V) then goto m", which uses V in a predicate. Examples of *c-use* include the assignment statement "V := X" and the output statement "print X", both of which use X computationally. A *dpu* set, denoted dpu(x,i), is the set of all (j,k) edges such that x is an element of p-use(j,k) and there exists a def-clear path with respect to x from i to j. A *dcu* set, denoted dcu(x,i), is the set of j nodes such that x is an element of c-use(j) and there exists a def-clear path with respect to x from i to j.

Additional testing criteria include: *all-p-uses* (for each definition, for each dpu there is a path tested); *all-p-uses/some-c-uses* (for each def, for each dpu there is a path tested; if for some def there is no dpu, choose some dcu); and *all-c-uses/some-p-uses* (for each def, for each dcu there is a path tested; if for some def there is no dcu, choose some dpu). The Rapps/Weyuker hierarchy of the relative power of the criteria is: all-nodes, all-edges, all-defs; all-uses, all-du-paths; all-p-uses; all-p-uses/some-c-uses; all-c-uses/some-p-uses.

All these testing criteria are interrelated, in that some imply others. For example, all-paths implies all-du-paths. A program graph may contain an infinite number of paths (if it contains a loop), but even graphs with loops contain a finite number of du-paths.

In summary, dataflow analysis may discover “interesting paths” to test, and reduce the cost of path testing. Dataflow analysis cannot guarantee better reliability than path testing.

19. Unit Testing: Partition Analysis

Reading: Richardson85, Hamlet88

[Guest lecturer: Debra Richardson, University of California, Irvine]

Partition analysis is an analysis method based on the specification as well as the implementation, and involving both static and dynamic analysis. The basic idea is to divide the input into a set of partitions, each of which might be studied exhaustively. The method partitions the input domain by analyzing the structure of both representations (specification and implementation). It produces the smallest domains that can be analyzed independently; testing and verification can be applied to each subdomain.

Symbolic evaluation provides common representations of specification and implementation. Procedures can be partitioned in two ways: the specification partitions the domain into subsets of input data that *should* be processed uniformly; the implementation partitions the domain into subsets that *actually are* processed uniformly.

Partition analysis testing provides a demonstration of the dynamic consistency between the specification and the implementation. For each procedure subdomain, test data is selected by computation testing criteria and domain testing criteria. The procedure partition decomposes structural and functional testing.

An experiment to evaluate the partition analysis method versus mutation analysis looked at 34 mutants of the same program. All known errors were found, all seeded errors (through mutation analysis) were found, and equivalence was demonstrated for most correct procedures (mutants that were semantically equivalent).

Some advantages of the method are that it can be applied to several different types of specification languages, it can be applied throughout the life cycle, it combines the benefits of testing and verification, and it results in tests based on both the specification and the implementation (and so addresses missing path errors and not just domain and computation errors).

[The lecture includes a detailed example of partitioning a prime number predicate function.]

20. Unit Testing: Coverage

Reading: DeMillo78, Hamlet77

[Guest lecturer: Dick Hamlet, Portland State University]

The deficiency in all variants of path testing is that although a path is traversed, the data used does not explore the possible state space for that path. Just as an untried path could harbor an arbitrary defect, so a path tried only with a few values could harbor an almost arbitrary defect. People (and symbolic-execution systems) often choose nonrepresentative data.

The idea of *data coverage* and *mutation testing* is, given a program P and a set of test data T, the test is (*mutation* or *data coverage*) *adequate* if and only if (1) T succeeds, so P is not shown to be incorrect, and (2) no change to P could be made without altering at least one test result. If any part of P can be changed without T detecting it, T is inadequate, because it fails to properly cover that part.

Each expression in a program is textually altered to form a number of complete program *mutants*, each arising from one change to one expression. All mutants and the original are executed on given test data. Any mutant whose output differs from the original is said to be *killed*. If at the end of testing any mutants remain alive, the changed expressions they contain correspond to expressions of the original program that have not been adequately covered.

Mutation can replace path analysis, but in practice it is easier to first cover paths, then alter expressions. Mutation is expensive, because even severely limited changes produce far too many alternatives. Some mutations fail to halt, and must remain alive or be arbitrarily eliminated. Equivalent programs may be difficult or impossible to detect.

[The lecture includes detailed examples of implementation of mutation testing, one due to Hamlet and one to DeMillo and Budd.]

21. Unit Testing: Limitations

Reading: Weyuker88

[Guest lecturer: Elaine Weyuker, Courant Institute, New York University]

Two significant problems concerning testing are that we don't use adequacy criteria for tests, and we don't know whether an adequacy criterion is any good. Axiomatization is an approach to solving these problems, because it provides a basis for the definition of new program-based adequacy criteria.

Fundamental questions to consider are the role of an adequacy criterion and the relationship between an adequately tested program and a correct program. A program may be correct and it may be adequately tested, but neither condition necessarily implies the other. The *correctness condition* is the requirement that P(t) = S(t) for every t in T, which means that the program computes the same function as the specification for all elements of the input domain.

Concepts of the axiomatic treatment of adequacy criteria include applicability (for every program there exists an adequate test set; for every program there exists a finite adequate test set), non-exhaustive applicability (there is a program P and a test set T such that P is adequately tested by T and T is not an exhaustive test set); monotonicity, inadequate empty set, antiextensionality, programs of the same shape, antidecomposition, Gödel numbering of programs, renaming, Gödel-class number, canonical form, and complexity of a test set.

[The lecture also includes a comparison of several popular testing criteria (black-box, path testing, etc.) in terms of the axioms discussed.]

22. System Testing: Integration and System Testing

Reading: Beizer84, Chapters 5-6

Some conclusions about unit testing are that no single method is universally successful, the cost of testing is an implicit constraint, and reliability is not the same as correctness. Other kinds of testing are also important. Integration testing assumes that unit testing was successful, and it focuses attention on interfaces among units. Interface problems include incorrect invocation syntax, incompatible types, and incorrect or incomplete specifications.

Integration strategies include top-down (test the higher level units first, incrementally integrating lower levels), bottom-up (inverse of top-down), and big-bang (test everything at once). The advantages of top-down integration are that it is based on stepwise refinement and it exposes control flow errors early; its disadvantages are that it requires stubs and that there may not be a clear "top." The advantages of bottom-up integration are that it allows more parallel activity of the testing team and it allows reuse more easily; its disadvantages are that it requires drivers and that system architecture flaws are discovered late. The advantage of big-bang integration is that it avoids problems that never arise; its disadvantage is that it avoids problems until it is too late.

Integration procedures must specify who corrects defects found during integration testing and how much retesting must be performed after each such correction.

System testing looks at the whole systems, not just the software. It includes system-level functionality testing, performance testing, and acceptance testing. The acceptance testing phase may include alpha testing (in-house) and beta testing (by customers).

Test plans are critical to the effectiveness of testing. An IEEE standard for test plans describes these major sections of the plan: test-plan identifier, introduction, test items, features to be tested, features not to be tested, approach, item pass/fail criteria, suspension criteria and resumption requirements, test deliverables (test design specification, test case specification, test procedure specification), testing tasks, environmental needs, responsibilities, staffing and training needs, schedule, risks and contingencies, and approvals.

A test design specification includes these sections: purpose, outline, test-design specification identifier, features to be tested, approach refinements, test identification, and feature pass/fail criteria. A test case specification includes these sections: purpose, outline, test-case specification identifier, test items, input specifications, output specifications, environmental needs, special procedural requirements, and intercase dependencies. A test procedure specification includes these sections: purpose, outline, test-procedure specification identifier, purpose (of this procedure), special requirements, log, execution, and contingencies.

23. System Testing: Performance Analysis

Reading: Bentley84, Bentley87

A "back of the envelope" analysis is sometimes useful to get a feel for a particular value. For example, when does overnight mail beat direct transmission? Assuming 24-hour delivery for overnight mail and 1200 baud direct transmission, a computation yields direct transmission is limited to 12,960,000 bytes per day. Larger items can be sent faster by overnight mail.

Analysis of algorithms can be used to identify expensive code, identify significant constants, and predict best and worst cases.

[The lecture includes discussion of heuristics for identifying best and worst cases from programming idioms. Sorting algorithms are used for examples.]

24. System Testing: Regression

Reading: Leung89, Beizer84, Chapter 8

[Guest lecturer: Liz Gensheimer, CONVEX Computer Corporation]

What is regression testing? The dictionary definition state: to regress is to return to a worse or more primitive state or condition. In the world of software, a regression represents a perceived loss of functionality between evolutionary versions of the same product. This view of regression has its roots in the assumption that all directed evolutionary changes in a software product are positive; they are enhancements or improvements. Any change not in a positive direction is therefore a regression.

Regression testing is the testing phase of the software life cycle and includes the following tasks: unit testing (the individual components are tested in isolation); integration testing (the individual components are put together to form the complete product); feature testing (major functional components are tested in isolation within the framework of the complete product); system testing (feature interactions are tested); performance testing (the product is tested against certain benchmarks, or performance criteria); regression testing (the product is tested against a known baseline).

The purpose of regression testing is to verify that the new version of a software product is no less correct than the previous version. The dependency of regression testing on the existence of at least one previous version of the product sets it apart from other testing tasks.

There are two basic regression methodologies currently used in industry. These methodologies differ in the composition of the tests that make up the regression test suite. Although both methodologies test for continued functional correctness, the type of the test used can reflect different attitudes toward product development. The two methodologies are *positive* regression testing and *negative* regression testing.

The regression test suite is derived from the complete functional test suite. Only test cases that are known to pass are included. The regression suite becomes a baseline for validating that future releases are functionally equivalent. Because the tests selected represent historically validated, defect-free, functionality, this method is termed "positive regression testing." The purpose of positive regression testing is to prove that the evolutionary version of the software product is at least as correct as any previous version. Positive regression testing makes no direct statement regarding any defects that may have been removed except to show that the corrective measures did not adversely affect the basic functionality of the product. It is not a requirement that only test cases representing defect-free functionality be included in a positive regression suite. Good testing practice would suggest that test cases that currently fail or that test suspected weakness in the product be included as well (this practice acknowledges that software products are generally not guaranteed to be defect-free).

A positive regression test suit is characterized by an emphasis on systemic testing (the product is tested as a complete entity rather than each feature being tested in isolation); tests represent commonly used paths; tests are pre-existing—they are derived from functional test suites (typically 20% of the available tests are used to test 80% of the code).

Some advantages of positive regression testing are the tests cover a majority of the commonly used paths; failures represent true defects (assuming that individual test cases are correct)—there should be no user bias; the product as a whole is tested rather than testing features in isolation. Some disadvantages are the test suite requires constant updating to include new functionality; the breadth of the test is limited by intimacy with the development effort; the tests do not cover all possible paths—the ones omitted are the least commonly used paths, which are in general also the less well test paths.

Many times a regression test suite will be constructed of test cases that represent failures reported by end users of the software product. Because each test case represents a true or perceived defect in the product, the use of such a suite is termed "negative regression testing." The purpose of negative regression testing is to verify that defects found in previous versions of the product have been removed. Negative regression testing does not verify that new defects have not been introduced during the defect removal.

A negative regression test suite is characterized by a wide variety of test cases representing a relatively small percentage of the code (defects tend to cluster rather than be evenly distributed); these are "real world" scenarios (failures occurring in user environments rather than artificial test environments). There are two types of tests in a negative regression test suite: tests that historically fail and tests that failed sometime in the past but now pass. A test case making a transition from fail to pass indicates that the defect has been removed.

Some advantages of negative regression testing are the tests provide evidence that defects have been corrected; the tests are additions to the existing functional test suite—this make the entire suite more complete. Some disadvantages are the tests represent failures only; uncommon, or least used, paths are tested rather than commonly used paths; user bias is introduced into the defect analysis. A very real danger in using a negative regression test suite is that failures are initially identified by end users All user reported failures should be carefully screened to reject instances of user error. Because the test suit is based on user reported failures, it will still be biased towards the users' perception of the product functionality.

25. System Testing: Performance Testing

Reading: Dongarra87

[Guest lecturer: Nelson Weiderman, Software Engineering Institute]

Software testing normally includes testing for functionality (does it do what it is supposed to do?), reliability (does it have bugs?), usability (does it behave in a friendly manner during installation, operation, and maintenance?), and performance (does it perform its function fast enough, and predictably?). Performance testing can be applied to processors, operating systems, applications programs, compilers, peripheral devices, networks, busses, memory hierarchies, and systems. Some measures of performance are *throughput* (work per unit time), *response time* (time per unit of work), *determinism* (regularity of response), and the static and dynamic size of the product. Software performance must be measured in the context of a system; the performance of the product is influenced by the hardware/software system environment, the development tools (compiler, linker, operating system), the algorithms used, and the coding techniques.

Benchmarks are used to measure both absolute performance and relative performance (relative to previous versions or to competing products). Performance should be measured at each level of abstraction; Dongarra's hierarchy of performance tests is: computer operations at the machine level, program kernels, basic routines or building blocks for applications, stripped-down programs, full-scale programs, and experimental techniques. Measurement tools include a stopwatch, software timers (provided by the programming language or the operating system), hardware timers, logic analyzers, and link maps.

The principles of benchmarking are to be sure you are measuring what you think you are measuring, assess the costs and benefits of benchmarking, and carefully control for sources of variation.

Sources of variation include systematic errors (effects that introduce a constant source of bias independent of the number of repetitions of the measurement); and statistical (random) errors (effects that introduce a variable source of bias depending on environmental influences). Characteristics of statistical errors include that they may occur frequently or rarely; they may have a large or small effect; and the frequency and magnitude are from known distributions or completely random. Note that measurements may have both systematic and random errors.

Examples of systematic sources of variation are fast or slow system clocks, incorrect "overhead" adjustments, memory alignment problems, memory hierarchies, and pipelined architectures. Good benchmarking technique must try to eliminate or compensate for such sources of variation. Examples of statistical sources of variation are clocks with poor resolution, interrupts from clocks, daemons, or user actions; and the sharing of resources in a multiprogramming or timesharing environment. Statistical techniques can be used to compensate for these effects; measurements are repeated until the statistical mean is within a given tolerance of the population mean.

Performance data must be analyzed to compare machines, compilers, etc. It is often desired to have one number representing relative system performance, such as *mips*, *kwips*, *mflops*, etc. The data analysis requires some statistical sophistication. It is necessary to draw attention to exceptional results, to provide a profile of a multi-faceted resource (strengths and weaknesses), and to produce meaningful numbers (such as knowing to use the geometric mean rather than the arithmetic mean to average normalized numbers).

Some performance issues for compilers are compile time vs. link time vs. run time; time vs. space; micro vs. macro language features; generated code vs. runtime system code; performance at different levels of stress; thwarting the optimizers; and performance of subsidiary components (debugger, program library system). Some "games" that compiler vendors play include modifying the benchmark, optimizing the benchmark program, suppressing checks, reporting selectively, tuning the configuration, tuning the compiler to the benchmark, judicious choices of compiler options, and judicious choices of operating system generation options.

In summary, performance testing is not for amateurs. The major issues are what to measure, how to measure, identifying and controlling sources of variation, and organizing and reporting results.

[The lecture also includes several examples of benchmarks.]

26. System Testing: Environments and Tools

Reading: Craddock87, Brown89

Several kinds of system testing activities can be supported by tools. Program proving activities include writing specifications, massaging assertions, creating assertions, and proving implications. Review activities include creating review materials, presenting, reviewing, recording, and reporting. Testing activities include generating tests, analyzing specifications, analyzing implementations, executing tests, comparing and analyzing results of tests.

Tools to assist program proving include verification condition generators, theorem provers, symbolic executors, and assertion annotation support. Review tools include document preparation systems, electronic mail, version control and configuration management tools, and demonstration equipment. Testing tools include static analyzers (such as syntax checkers, semantics checkers, and dataflow analyzers), dynamic analyzers (such as profilers, monitors, simulators, and debuggers), test data generators, file comparators, and execution script support.

Some examples of environments for some of these activities include the Cornell Program Synthesizer and Turbo Pascal for debugging support; Apollo's DSEE for support for versions and update procedures; and DEC's VAXset for support for regression test suite management.

Current research is aimed at developing more specification languages and more powerful theorem provers to assist in program proving, and systems to support mutation and coverage testing, random testing, and hybrid testing methods.

27. System Testing: Safety

Reading: Leveson86

Software safety issues can be exemplified by the case of the Therac-25 therapeutic radiation machine. The machine evolved from earlier designs, with hardware function being replaced by software. Software faults contributed to accidents resulting in at least 2 deaths.

Contributions to software safety can be made throughout the development of a product, including requirements analysis, specification and design, implementation, verification and validation, and operation and maintenance. A method is *hazard analysis*, which includes a preliminary risk assessment; identification of potential subsystem hazards as the design evolves, especially those related to component failure; detailed analysis of the design of the system, especially considering interfaces and total system failure; and an analysis of procedures for operating and supporting the system.

Hazard analysis techniques include fault tree analysis (a search for causes of hazards), event tree analysis (examination of the consequences of events), and simulation (using models to predict behavior). Fault tree analysis is performed after preliminary hazard analysis has identified the important hazards. It is a backward trace through the system to find all conditions that ultimately cause a hazard. It often is a mix of software fault tree analysis and hardware fault tree analysis. It is effective when hazards are clearly identified, and it may be used to identify critical components. The system may be repaired to prevent hazardous conditions.

Safety and reliability are not the same. Reliability is freedom from failure, and can be measured as the number of faults per unit time. Safety is freedom from mishaps, where the risk is less than some “magic number” that is a function of hazard probability and hazard criticality.

Time Petri net analysis can be conducted after preliminary hazard analysis and preliminary system design. It is a reachability analysis of a Petri net model to discover hazardous states.

Some fundamental concepts of software safety are *fault-tolerant* (the system continues to provide full functionality and performance in spite of faults), *fail-soft* (the system continues to operate, but at reduced performance and/or functionality), and *fail-safe* (damage is limited, but no guarantees of operations). Two *locking* concepts are *interlock* (force the correct sequence of events by preventing an event from happening too early) and *critical section* (prevent an event from occurring while a condition holds). Recovery possibilities include *backward* (return to a previous state and try again), *forward* (repair the current state and continue), and *shutdown* (abort the current process).

Procedures and regulations related to safety include licensing, which is regulation of operation of software systems, and standards, which address regulation of creation. For example, British Ministry of Defence draft standards MOD 00-55 and 00-56 address procedures for safety-critical software. They require the use of formal methods in software creation, and they forbid some “unsafe” practices including floating point arithmetic, recursion, assembly level programming languages, and interrupts. They require testing legal values (such as with boundary value analysis) and illegal values and the use of randomly generated test cases.

Software safety is difficult because safety is a total system problem (including people), there are multiple causes of hazards, and risk assessment involves personal values.

28. System Testing: Metrics and Reliability

Reading: Goel85, Beizer84, Chapters 9-10

[Guest lecturer: John Musa, AT&T Bell Labs]

Fundamental terminology for discussions of reliability includes the following. A *failure* is a departure of program operation from the user requirements (customer needs); it is a customer-oriented concept. An example of a failure metric is 3 failures/1000 CPU hour. A *fault* is a defect in a program that causes failure; it is a developer-oriented concept. An example of a fault metric is 6 faults/1000 source lines. Failure *intensity* is the number of failures in a time period; an example of a failure intensity metric is 25 failures/1000 CPU hr. *Reliability* is the probability of failure-free operation for specified time; for example, 0.82 for 8 CPU hour day.

Software reliability is affected by fault introduction, fault removal, and the operational profile of the software. Reliability models focus on fault removal, and they assume that decreasing failure intensity is essentially a random process. Execution time models may be based on actual execution time or on calendar time. Decreasing failure intensity can be modeled with a basic distribution or a logarithmic Poisson distribution. The models use a level parameter and a decay rate parameter.

An advantage of the models is their predictive capability; a model allows prediction of the amount of additional execution time required to reach a particular failure intensity objective. The calendar time to execution time ratio is determined, based on resource limitations that affect the pace of operations. Resource limitations might include failure correction personnel, failure identification personnel, and computer time. The current quality status of the software can be determined by recording the execution times of failures, and then running a reliability estimation program based on the reliability model. Given a failure intensity objective, this program can the predict when the software is of a quality satisfactory for release.

Reliability models can be used in other ways, such as evaluating the effect of new software engineering technologies. It is useful to answer questions such as, with the failure intensity objective and other variables constant, how much does the new technique decrease the amount of testing required? One experiment showed that design inspections reduced testing by 17%. The models can also assist in getting the needed software quality level, and in maintaining service standards while providing new features. They can help increase software productivity by helping to define customer needs precisely, giving managers more quantitative information for making decisions, and for controlling development with earlier identification of problems.

29. Final Review

30. Final Examination

Bibliography

Beizer83	Beizer, Boris. *Software Testing Techniques*. New York: Van Nostrand Reinhold, 1983. Second edition published 1990.
Beizer84	Beizer, Boris. *Software System Testing and Quality Assurance.* New York: Van Nostrand Reinhold, 1984.
Bentley84	Bentley, Jon. “Programming Pearls: Back of the Envelope.” *Comm. ACM 27*, 3 (Mar. 1984), 180-184.
Bentley87	Bentley, Jon. “Programming Pearls: Profilers.” *Comm. ACM 30*, 7 (July 1987), 587-592.
Brown89	Brown, John M. and Gilg, Thomas J. “Sharing Testing Responsibilities in the Starbase/X11 Merge System.” *Hewlett-Packard Journal* (Dec. 1989), 42-46.

Craddock87	Craddock, Linda L. "Automating a Software Development Environment." *Proc. Digital Equipment Corporation Users Society.* DECUS, Dec. 1987, 147-153.
DeMillo78	DeMillo, Richard A., Lipton, Richard J., and Sayward, Frederick G. "Hints on Test Data Selection: Help for the Practicing Programmer." *IEEE Computer 11*, 4 (Apr. 1978), 321-328.
Dongarra87	Dongarra, Jack, Martin, Joanne L., and Worlton, Jack. "Computer Benchmarking: Paths and Pitfalls." *IEEE Spectrum 24*, 7 (July 1987), 38-43.
Dyer87	Dyer, Michael. "A Formal Approach to Software Error Removal." *J. Systems and Software 7*, 2 (June 1987), 109-114.
Fagan76	Fagan, M. E. "Design and Code Inspections to Reduce Errors in Program Development." *IBM Systems Journal 15*, 3 (1976), 182-211.
Goel85	Goel, Amrit L. "Software Reliability Models: Assumptions, Limitations, and Applicability." *IEEE Trans. Software Engineering SE-11*, 12 (Dec. 1985), 1411-1423.
Gries81	Gries, David. *The Science of Programming.* New York: Springer-Verlag, 1981.
Hamlet77	Hamlet, Richard G. "Testing Programs with the Aid of a Compiler." *IEEE Trans. Software Engineering SE-3,* 4 (July 1977), 279-290.
Hamlet88	Hamlet, Richard G. and Taylor, Ross. "Partition Testing Does Not Inspire Confidence." *Second Workshop on Software Testing, Verification, and Analysis.* IEEE Computer Society Press, July 1988, 206-215.
Hantler76	Hantler, Sidney L. and King, James C. "An Introduction to Proving the Correctness of Programs." *ACM Computing Surveys 8*, 3 (Sept. 1976), 331-353.
Hoare69	Hoare, C. A. R. "An Axiomatic Basis for Computer Programming." *Comm. ACM 12*, 10 (Oct. 1969), 576-583.
Howden76	Howden, William E. "Reliability of the Path Analysis Testing Strategy." *IEEE Trans. Software Engineering SE-2*, 3 (Sept. 1976), 37-44.
Kemmerer85	Kemmerer, Richard A. and Eckmann, Steven T. "UNISEX: A UNIx-based Symbolic EXecutor for Pascal." *Software—Practice and Experience 15*, 6 (May 1985), 439-458.
Leung89	Leung, Hareton K. N. and White, Lee. "A Study of Regression Testing." *Sixth International Conference on Software Testing.* U.S. Professional Development Institute, 1989.

Leveson86	Leveson, Nancy G. "Software Safety: What Why, and How." *ACM Computing Surveys 18*, 2 (June 1986), 125-163.
Linger88	Linger, R. and Mills, Harlan D. "A Case Study in Cleanroom Software Engineering: The IBM COBOL Structuring Facility." *Proc. OMPSA 88*. IEEE, 1988, 10-17.
Mills86	Mills, Harlan D. "Structured Programming: Retrospect and Prospect." *IEEE Software 3*, 6 (Nov. 1986), 58-66.
Myers79	Myers, Glenford J. *The Art of Software Testing*. New York: John Wiley & Sons, 1979.
Rapps85	Rapps, Sandra and Weyuker, Elaine J. "Selecting Software Test Data Using Data Flow Information." *IEEE Trans. Software Engineering SE-11*, 4 (Apr. 1985), 367-375.
Richardson85	Richardson, Debra J. and Clarke, Lori A. "Partition Analysis: A Method Combining Testing and Verification." *IEEE Trans. Software Engineering SE-11*, 12 (Dec. 1985), 1477-1490.
Selby87	Selby, Richard W., Basili, Victor R., and Baker, F. Terry. "Cleanroom Software Development: An Empirical Evaluation." *IEEE Trans. Software Engineering SE-13*, 9 (Sept. 1987), 1027-1037.
Weyuker88	Weyuker, E. J. "The Evaluation of Program-Based Software Test Data Adequacy Criteria." *Comm. ACM 31*, 6 (June 1988), 668-675.
Wienberg84	Wienberg, Gerald M. and Freedman, Daniel P. "Reviews, Walkthroughs, and Inspections." *IEEE Trans. Software Engineering SE-10*, 1 (Jan. 1984), 68-72.

3.6. Software Project Management

Students' Prerequisites

- computer science background
- reasonable knowledge of programming
- some experience of team software development
- some exposure to the software development *process*
- industrial experience is an advantage

Objectives

- The student will understand the requirements for the content of a project management plan.
- The student will be able to write a plan for a small project according to an established standard.
- The student will understand the role of the manager in each phase of the software development life cycle.
- The student will appreciate the key roles managers play in software development efforts.
- The student will appreciate economic and customer-driven factors and their role in the eventual form of the software product.

Philosophy of the Course

Software project management remains different from project management in other, more established fields for a number of reasons: software is a *brain product* only, unconstrained by the laws of physics or by the limits of manufacturing processes. It is difficult to detect and prevent defects in software. Software can be highly complex. Finally, as a discipline, software development is so young that measurable, effective techniques are not yet available, and those that are available are not well-calibrated. Despite these difficulties, there is an increasing body of knowledge about software project management. This course presents that knowledge and also points to promising new conceptual material.

Because of the level of experience of typical graduate students and their probable career paths, the course emphasizes methods, tools, and techniques sufficient to run small projects of 15 persons or fewer. Students may eventually run larger groups, consisting of teams of such teams, so some discussion of scaling up is included. Also, information about assessing the efficacy of the software development process is presented.

Software project management should be part of software engineering programs because the technology of developing software is so closely tied to the techniques of management. What is taught here is first-line management; product management and higher levels of management are the domains of schools of business. Some of the material discussed in the course, such as scheduling tools like PERT and concepts like tracking progress, is also part of project management courses offered by business schools; but the point of this course is to consider such tools and concepts in the context of software development.

Syllabus

The syllabus assumes 29 class meetings, including midterm and final examinations. Each meeting is planned to accommodate both lecture and class discussion. In addition, two special lectures are included in the videotape package and may be used at the instructor's discretion.

The course has four major segments:

- Estimating (What does it take to do the job?): classes 2-6
- Planning (How will the job be done?): classes 7-13
- Process Management (Executing the plan): classes 15-22
- Special Topics and Circumstances: classes 23-28

1. The Nature of Software Production
2. What Needs to be Estimated
3. Understanding Software Costs
4. Estimation Models I: COCOMO
5. Estimation Models II: Function Points
6. Risk Determination: Fallacies in Estimations
7. Work Products; Using Standards
8. Determining and Using Objectives and Milestones
9. Risk Management: The Use of Prototypes
10. Team and Subteam Organization
11. Resource Acquisition, Allocation, Training
12. Contents of a Project Plan
13. Discussion of Results of Estimation Exercise
14. Midterm Examination
15. The Role of Configuration Management
16. Implementing Configuration Management
17. The Role of Quality Assurance
18. Implementing Quality Assurance
19. Tracking, Reviewing, Adjusting Goals, Reporting
20. People Management I: Managing Yourself
21. People Management II: Analysts, Designers, and Programmers
22. People Management III: Testers, Support Staff, Customers
23. Assessment
24. Managing Sustaining Engineering
25. Legal Issues I: Patentability and Copyright
26. Legal Issues II: Liability and Warranty
27. Special Cases
28. Assessing the Organization's Ability to do the Process
29. Final Examination

Special Lecture 1. Causal Analysis: A Defect Prevention Process
Special Lecture 2. Enhancing Professionalism

Summaries of Lectures

1. The Nature of Software Production

The development of software products is different from the development of other kinds of products, especially with respect to manufacturing differences and item volume. Understanding the differences is important to software engineers. A critical aspect of software engineering is management. Software engineering management is sufficiently different from other kinds of management, including other engineering management, to warrant detailed study by software engineers. The manager has a key role in decreasing cost and improving productivity. Natural talent is a factor in management performance.

The course is organized as four major areas: estimating, planning, managing, and special topics. The issues in software estimating are determining what needs to be estimated, understanding the factors that affect software costs, estimation models, and fallacies in estimation. Planning issues are identifying work products, standards, objectives, milestones, risk management, team organization, resource acquisition, resource allocation, training, and documenting the project plan. Issues in process management are configuration management, quality assurance, tracking, reviewing, adjusting goals, reporting, and people management. Special topics include assessment; managing sustaining engineering (software maintenance); legal issues such as copyright, liability, and warrantees; taking over a project already underway; recovering when the project is failing; and evaluating the software production organization.

Students are assigned the task of writing a short essay on the question, "What makes it difficult to tell how long it will take to make a software product?" The essay not only sets the stage for the next class, but gives the instructor feedback on the students' writing ability.

2. What Needs to be Estimated

Reading: Brooks75, Chapters 1, 8, 9

Software cost estimation is difficult for many reasons: software products are often unlike any that the development team has ever built before; the project usually starts with incomplete specifications; the specifications change throughout the development effort; skills and talents of individuals affect their contribution but are difficult to predict; and the relationship between the size and the complexity of the code is hard to establish.

The behavior of the development team is an especially difficult thing to predict. Studies have shown that two individuals with similar education and experience may have a 7 to 1 ratio of productivity. Small teams may be more productive than large teams, due in part to the reduced number of communication paths and the ability of the team's "stars" to be more dominant. Software development has exacting demands that reduce productivity.

The major factors affecting software cost are product size, production time, complexity, effort, and resources. A software manager must develop an ability to estimate each of these.

3. Understanding Software Costs

Reading: Boehm87

The effects of software backlogs are that programs are rushed to completion due to pressure to “get on to the next job.” The backlog in the data processing industry is about five years.

Two categories of ways to understand software costs are influence-function (black box) approaches and cost-distribution (glass box) approaches. The former include tools and methods, people quality, hardware, and the software process; the latter includes experience vs. new hires, upgrading the environment, labor vs. capital, and code vs. documents. Essentially the black-box approaches assume that software costs can be managed by concentrating on the structure of the development process. Glass-box approaches balance the set of opposing factors listed to manage costs.

Factors that affect productivity include volatility of requirements, the degree of reliability required in the product, the use of modern programming practices, the complexity of the product, and the capability of the personnel and the team. Managers need to understand the degree to which each of these factors can affect productivity; experience has shown that the five factors above are listed in order of increasing importance.

Productivity improvement opportunities include making people more effective, making development steps more efficient, eliminating steps, eliminating rework, building simpler products, and reusing previously built components.

4. Estimation Models I: COCOMO

Reading: Boehm81

COCOMO (COnstructive COst MOdel) is a cost estimation model developed by Boehm at TRW in the 1970s. The model can be used at three levels of detail: basic, intermediate, and detailed. It can be applied for three different kinds of product: organic, semi-detached, embedded, where organic is primarily for internal use in a company, semi-detached is a product developed according to some standard, and embedded is usually a real-time system. The model assumes that the software requirements are relatively stable (low volatility), that the organization uses good software engineering practice, and that the development team is well managed.

The model is used to predict effort measured in person-months as an exponential function of the number of delivered source instructions (a term subject to many interpretations). In the basic model, the exponent in the function varies from 1.05 to 1.20 depending on the product. The model also is used to predict development time as an exponential function of the effort; the exponent ranges from 0.32 to 0.38 depending on the product.

The model was developed by fitting a curve (the function) to data collected over a number of projects. The coefficients and exponents were further validated with other project data. Different organizations with different development environments will need to modify these values based on their own past experiences. In particular, development in Ada is being studied to further refine the model.

The estimate of the size of the software product is the critical factor in the model. Size estimation can be done in a variety of ways, including by analogy with similar systems, wideband Delphi techniques, Putnam's model, linguistic models, and function points.

5. Estimation Models II: Function Points

Reading: Albrecht83, Symon88

The wideband Delphi technique is used to achieve some consensus of the developers on the estimated size of the software product. It consists of the initial steps of having developers examine the requirements, discuss the requirements as a group, and then independently and anonymously estimate the size of the product. Then the following steps are repeated until some degree of convergence is achieved: the mean of the estimates is computed and compared to the individual estimates, the results are discussed, and new anonymous estimates are made.

Putnam's model tries to temper estimates by adding four times the best guess of lines of code to the highest and lowest imaginable estimates, then dividing by six.

Function points are computed as the sum of several functional factors times a term involving weighting factors. The functional factors are the number of user inputs, the number of user outputs, the number of user inquiries, the number of files, and the number of external interfaces. The weighting factors include whether or not the system requires reliable backup and recovery capability; data communications is required; there are distributed processing functions; performance is critical; the system will run in an existing heavily utilized operational environment; the system will require on-line data entry; on-line data entry requires input transactions to be built over multiple screens or operations; the master files are updated on-line; the inputs, outputs, files, or inquiries are complex; the internal processing is complex; the code is designed to be reusable; conversion and installation are included in the design; the system is designed for multiple installations in different organizations; the application is designed to facilitate change and ease of use. Other function point metrics are productivity, quality, cost, and documentation size. Studies have shown a relationship between function points and lines of source code in various languages (examples: Ada 72, Fortran 105, C 128).

Advantages of effort estimation schemes include: aiding schedule generation, reasonableness, and simplicity. Disadvantages include they are too believable, too dependent on one variable, and subjective. The advantages of function points are that they are based on a deeper understanding of software complexity, involve more than one variable, include looking at requirements, and can involve the user. The disadvantages are that function point analysis is more complex, subjective, and biased toward data processing applications.

[The lecture also introduces a student assignment: For the ITRAC (a software system for which the students are supplied a requirements document), calculate function points, size, person-months, time of development, number of full time personnel, and person loading at 1/3 and 2/3 of the way through development.]

6. Risk Determination: Fallacies in Estimations

Reading: Brooks75, Chapter 2, Abdel-Hamid86, Kemerer87

Estimation techniques are far from perfect. Software managers must understand the risks associated with poor estimates. Studies indicate that requirements changes account for 85% of cost increases, while poor estimates account for only 15%. Choosing a software contractor on the basis of cost estimates is risky. Estimates fail because of undue optimism, confusion between effort and progress, gutless estimating, poor progress monitoring, and adding staff late in a project.

Rules of thumb for distribution effort include Brooks': 1/3 planning, 1/6 coding, 1/4 unit test and early integration test, 1/4 integration test. Tomayko's rule of thumb for Ada development: 1/2 planning, 1/12 coding, 1/4 unit test and early system test, 1/6 integration test.

Which development phase is the most dangerous to underestimate? Testing, because it is too late to accommodate schedule adjustments.

Different estimates make different projects: Abdel-Hamid has found some estimates to be self-fulfilling prophecies.

Kemerer has surveyed several projects to attempt to validate the COCOMO and function point models. He found 600% overestimates of effort using the COCOMO model and 38% low estimates of software size using function points. The conclusion is that current estimation methods can not be applied blindly, and that fine-tuning to a particular company and development environment is necessary.

7. Work Products; Using Standards

Reading: DoD88

Typical work products in a poorly managed software project include only requirements, and code. Standards, which suggest other documentation and their contents, may be external (2167, NASA, IEEE), internal (such as the Boeing Embedded System Standard), or project. Standards may specify process criteria and documentation criteria. Standards affect development in many ways, including improving communication. The developers must allow time for producing the deliverables required by the standard. Tools can help. Some standards are overkill for some kinds of projects. This can often be overcome with appropriate tailoring.

Examples of unique work products from in-house software tool development are specifications, source code, user manual, tests, maintenance guide, and porting information. Examples from a commercial batch-type system are an installation guide and site dependencies. Examples from an embedded system delivered to the government are memory maps and interface control documents.

8. Determining and Using Objectives and Milestones

Reading: Cori85

Reducing costs by not planning results in cost increases. Effective schedules have many characteristics, such as being understandable, detailed, clear as to critical tasks, flexible, reliable, cognizant of resources, compatible with competing plans.

Seven steps to planning are: define requirements, create the work breakdown structure, define the sequence of activities, estimate the length of activities, adjust the schedule to the time constraints, adjust the schedule to the resource constraints, and then review the plan.

Milestones are used to create intermediate review points with specific documents associated with them.

Milestone tracking can be aided by graphical representations. Gantt charts are good for showing concurrent activities and time for each activity. They are useful for projects with a small number of activities, and they can be derived from activity networks. PERT charts are activity networks that help identify dependencies. The critical path is the longest path through the network in terms of the total duration of tasks. In complicated projects many "near-critical" tasks and paths may exist. Delays in a non-critical path task may result in a new critical path. Lengthening the critical path lengthens the project. Critical path analysis can help determine the minimum elapsed time to complete the project, which tasks determine whether the project is completed in the minimum time, and the latest time a particular activity can be begun without changing the overall project completion time.

[Student assignment: Prepare a PERT chart and derived Gantt chart for the ITRAC project using estimates developed during the previous assignment.]

9. Risk Management: The Use of Prototypes

Reading: Brooks75, Chapter 11, Boehm84, Boehm88

Risk reduction is part of a product life cycle; risk management is part of software project management. Product risk reduction activities include system test, reviews, and inspections. Another important way to reduce risks is through prototypes. This can allow early user feedback on interfaces or usability, test technical concepts, and clarify requirements. A prototype can also be a sales tool. Boehm's prototyping experiment suggests that prototyping leads to less effort and code, higher user satisfaction, and more maintainable products.

The spiral or "risk-driven" life cycle is an iterative process. At each stage, it is necessary to identify objectives, examine alternative implementations, and determine constraints on those alternatives. The model defers elaboration of low-risk elements, allows prototyping at any stage, and accommodates rework. Its major advantages include the facts that it focuses on reuse, allows for change, eliminates errors early, uses only appropriate resources, and approaches development and maintenance similarly. Its disadvantages are that it relies on risk assessment expertise and that it is not easily adapted to a project that requires compliance with military standard 2167A.

A risk management plan should be developed for a software project. The project manager should identify the top ten risk items, develop a plan to deal with each, review the list of risks monthly, highlight the status of risk items in overall project reviews, and take corrective action where necessary.

10. Team and Subteam Organization

Reading: Brooks75, Chapters 3 and 7, Mills83

A software manager must build appropriate teams. Issues include division of labor and specialization of function. Large organizational structures can be functional, project-oriented, or a matrix. Some software team structures include democratic, chief programmer, and "surgical." A democratic team works by consensus; team members are equals and vote on decisions. A chief programmer team has a chief, an assistant programmer, and a librarian. The surgical team is actually a scaling up of Harlan Mills' three-person chief programmer team. The surgical team has members with very different and clearly defined roles, with the basic objective of letting the chief programmer do the main design work, with the supporting cast filling in the other roles. Successful organizations are designed around the people available; this is the challenge to the manager.

11. Resource Acquisition, Allocation, Training

Reading: Dart87, Brooks75, Chapter 12, Cupello88

The resources needed for software projects come in several categories. Software tools include compilers or translators, editors, document processors, debuggers, specification and design tools, estimation tools, configuration management tools, system builders, and communications tools such as electronic mail.

Environments come in categories such as language-centered (Lisp machines, Rational Ada environments), method centered (Cadre's Teamwork), and toolkit (Unix).

Training issues include deciding when to provide it, how much to provide, and its structure.

12. Contents of a Project Plan

Reading: Brooks75, Chapter 10, Fairley 86, NASA86a

Brooks identifies these key documents as part of the project plan: objectives, specification, schedule, budget, organization chart, space allocation, and estimate-forecast-price (the balance that determines the eventual profit margin).

Fairley suggests a structure for a project plan with five components. The introduction includes a project overview, list of project deliverables, evolution of the plan, list of reference materials, and definitions and acronyms. The project organization section includes the process model, the organizational structure, the organizational interfaces, and the project responsibilities. The managerial process section describes management objectives and priorities; assumptions, dependencies, and constraints; risk management plan; monitoring and controlling mechanisms; and staffing plan. The technical process describes methods, tools, and techniques; software documentation; and project support functions. The section on work elements, schedule, and budget includes work packages, dependencies, resource requirements, budget and resource allocation, and a detailed schedule.

NASA has published a standard for a software project plan organized as ten parts. The introduction includes sections for identification, scope, purpose, organization, objectives, program constraints, program software schedules, and program controls. An applicable documents section lists reference documents, information documents, and parent documentation. The resources and organization section describes project resources (contractor facilities; government-furnished equipment, software, and services; and personnel), responsibilities, panels (review and advisory), and software development (organizational structure, personnel, resources, tools, techniques, methodologies, and the software environment). The life cycle management section includes concept and project definition, initiation, requirements definition, preliminary design, detailed design, implementation, software and system integration and testing, acceptance testing, and sustaining engineering. Management controls includes engineering master schedules and risk management (activities, activity network), reviews and reporting policies, risk management (high risk areas, technology risks, and disaster risks and recovery), and status and problem reports. The software support environment section addresses software development, software acquisition, software integration, operation and maintenance, and software tools. The software product assurance section describes configuration management, independent verification and validation, security, product assurance, interface definition and control, and waivers to policies and procedures (permanent, temporary, tool and testbed). The plan concludes with notes, appendices, and a glossary.

[Student assignment: Project Plan Assignment: Using either the appropriate DoD2167A DID, the NASA DID, or Fairley's outline, prepare a Project Management Plan for the ITRAC development.]

13. Discussion of Results of Estimation Exercise

Results of the estimation exercise from previous offerings of the course show considerable variation among students. Function points are made consistent by consistently interpreting the rules as stated by Albrecht. COCOMO is not very accurate for a project as small as the example (ITRAC). Wideband Delphi technique should have a moderator. Putnam's model strengthens estimation by analogy.

14. Midterm Examination

15. The Role of Configuration Management

Reading: Bersoff84

Software product integrity depends on the combined effects of three categories of disciplines: development, management, and control. The development disciplines include the common life cycle activities of specification, design, implementation, and testing. Management disciplines include the usual activities of planning, estimating, tracking, team-building, etc. The controlling disciplines are software quality assurance, software configuration management, and independent verification and validation. A commitment to configuration management by the entire organization is the key to its success.

Configuration management is responsible for maintaining the integrity of configuration items, evaluating and controlling changes, and making the product visible. Typical configuration items are: requirements, specifications, design documents, source code, object code, load modules, memory maps, tools, system descriptions, test plans, test suites, user manuals, maintenance manuals, and interface control documents. Two major types of changes reports are discrepancy reports (requirements errors, development errors, violations of standards) and change requests (unimplementable requirements, enhancements). The functions of the configuration management library are software part naming, configuration item maintenance and archiving, version control, and preparation of a product for release.

16. Implementing Configuration Management

Reading: Harvey86

A major part of the configuration management organization is the configuration control board (CCB). The principles guiding the CCB are the principle of authority, the principle of solitary responsibility, and the principle of specificity. Factors determining the CCB characteristics include hierarchical structures (having several boards for major subsystems of the overall system), scope, and composition.

Key factors considered by the CCB in evaluating proposed changes to the product include: size, complexity, CPU and memory impact, cost, test requirements, date needed, criticality of the area involved, resources available (skills, hardware, system), impact on current and subsequent work, approved changes already in progress, and politics (customer or marketing desires). An overriding consideration for all proposed changes is "is there an alternative?"

A typical process for evaluating a discrepancy report is: submit the report; log it; conduct a CCB evaluation of the report; if approved, then the development group makes the change; all affected configuration items are updated; the change closure is audited and logged. The CCB may also grant a waiver (allowing the discrepancy to remain), which is also audited and logged. A typical process for evaluating a change request is: submit the request; log it; conduct a CCB evaluation of the request; if authorized, the development group makes the change; all affected configuration items are updated, and the change closure is audited and logged. If the request is not authorized, that also constitutes change closure to be audited and logged.

Configuration management helps solve the "simultaneous update" problem, in which two or more developers work simultaneously on independent changes; both start from the same version of the code but the last one to save changes overwrites the changes saved earlier by others.

Version control is the management of the various releases of the product and the intermediate developmental versions between releases. These are often displayed graphically as a version tree. Tools for version control include sccs and rcs on UNIX systems, CMS, CCC, and DOMAIN. System descriptions identify which versions of which components constitute a particular version of a system. Tools for system description include simple text files, make, MSS, and DOMAIN.

A configuration management organization is responsible for developing, maintaining, and executing a written configuration management plan. Standards for such plans include: ANSI/IEEE 828-1983 and Department of Defense standards MIL-STD-483A (Air Force) and DoD-STD-2167A. A trend analysis of previous projects can assist in configuration management planning; typical trends are a decreasing number of change requests during the months following a release, but a number of discrepancy reports that starts low, increases for several months, and then declines.

[Note: The amount of material in this lecture was sufficiently large that some of it was actually presented at the start of the class period for lecture 17.]

17. The Role of Quality Assurance

Reading: Buckley84

How is quality exemplified in “hard” products? How is quality embodied in “hard” products? How is quality exemplified in software? How is quality embodied in software?

Software quality assurance grows out of quality assurance in other engineering disciplines, borrowing the body of experience, staffing, and tools.

18. Implementing Quality Assurance

Reading: Weinberg84; Basili87; Brooks75, Chapters 4, 5, 13

Software quality assurance is both a function performed in a software development organization and a separate body within that organization. The separation is necessary because engineers often lack the ability to distance themselves from development so as to observe possible discrepancies. A separate quality assurance (QA) organization is more likely to find errors and report poor engineering practice.

Management receives a variety of information from both inside and outside the development process; results of tests, inspections, and reviews come from the developers, while information on staff turnover, schedule slippages, and product failures come from outside. Models of software quality assurance include “full service” (which treats the product the way an ignorant customer would , and also participates fully in the process), “lip service” (which merely signs off on existing testing done by developers), consulting, and project-specific. Techniques include technical reviews, walkthroughs, and inspections of all the configuration items. Walkthroughs can often reveal the lack of “problems solved,” high-level logic errors, and dependency errors; they often fail to reveal missed details or missing requirements, due to the relative informality in how they are conducted. Inspections reveal details, but they can be too low level, so that some of the larger issues such as the architecture of the overall system are not reviewable.

Recent trends are toward quality-based software development processes. An example is the “cleanroom” technique. Its key elements are the use of formal methods for specification and design, implementors that do not execute their code, and statistically-based independent testing. The statistical approach to testing is to develop “normal” test cases, choose a random subset of the cases, test the code on these cases, and then inform the implementors that the tests passed or failed but not what errors were found. This approach overcomes the problem of prejudiced results when implementors tested their own code. The cleanroom approach has resulted in substantially fewer defects in delivered code.

19. Tracking, Reviewing, Adjusting Goals, Reporting

Reading: Brooks75, Chapters 6, 7, 14; Bernstein81; Crawford85

Software projects rarely fail because of large disasters, but because of a continuing series of small problems and schedule slippages. It is important to solve small problems before they become large ones. Project managers must maintain visibility of the process through written status reports, regularly scheduled review meetings, use of review boards, and use of audits. Status reports describe work accomplished since the previous report and estimates of the percentage of the work completed. Review meetings concentrate on status vs. action; in the reviews, problems are identified, not solved on the spot. Independent audits can involve project managers and technical staff to examine the development process, staff, products, quality assurance, and configuration management. Audits are not negative; they should be scheduled even if a project is not in trouble.

Project communication is a critical factor in tracking. Methods include "the manual," which is Brooks' concept of a comprehensive document such as the *System 360 Principles of Operation,* which guides the entire development, telephone logs, electronic mail and bulletin boards.

20. People Management I: Managing Yourself

Reading: Metzger87, Chapter 1

The manager must set the tone for the project: fear vs. leadership, don't hog credit, praise in public but criticize in private, stay technically competent, invite criticism and comment, fix your mistakes immediately, reward technical people the same as managers, instill the idea of service. The manager has a role as teacher for new staff members and for himself or herself. The manager should help project staff move ahead in their careers. The manager is a communicator: writing job descriptions, conducting project meetings, and effectively using the telephone, memos, and letters. The manager is a "historian" and should learn from mistakes and successes. The manager be able to address issues of performance evaluations and salary determination. The manager must pay attention to sources of potential conflict.

21. People Management II: Analysts, Designers, and Programmers

Reading: Metzger87, Chapters 2-4

What is the analyst's job? How do analysts come to understand the customer's problem? It is important to write down not only what is to be in the product but also what is not to be in the product. Risk reduction with analysts is improved by getting incremental approval, using analysts with "people" skills, and using teams.

Designers are the bridge between analysts and programmers. Design quality is difficult to judge, especially judging when a design is "good enough."

Avoid the programmers that act like prima donnas. Make an orientation schedule for new hires that is meaningful and helps them reach the point that they are making a contribution to the company.

22. People Management III: Testers, Support Staff, and Customers

Reading: Metzger87, Chapters 5-8

What are the personal characteristics of good testers? Managers can make testing better: review unit test plans—even if the reviews are informal; never drop acceptance-type tests; treat user documents as testable items; keep problems in perspective; accept necessary restarts.

What sort of support staff are needed and where are they likely to come from. To get the most out of support staff, the manager should recognize their contributions and retrain them as needed for additional or new responsibilities.

The manager should ensure that the organization is faithful to the customer. Don't lie to the customer and don't move on before the first product is finished.

23. Assessment

Reading: NASA86b

How are post-mortems, histories, lessons-learned documents, etc. useful? What makes an organization reluctant to share lessons learned?

A lessons-learned document typically contains: reference documents, parent documentation, project description (including organization and delivered items), assessments, and project evaluations. The project description includes the structure, key personnel, actual milestones reached, effort totals, and work products delivered. The assessments cover methods, practices and standards; unique concepts used; effectiveness of the development plan; specification quality (measured against such things as frequency of change requests and discrepancy reports); design, code, and test quality; personnel adequacy and effectiveness; and unusual problems and solutions. The project evaluation describes what worked and did not work in the actual product; what worked and did not work in the process; what can be done again; and what should never be done again.

24. Managing Sustaining Engineering

Reading: Collofello87

A variety of terms have been used to describe this part of the software process: maintenance, post-deployment software support, sustaining engineering, and software evolution. Some characteristics of software maintenance are that software is usually poorly designed for change; software degrades under maintenance, hardware does not; hardware has idiosyncratic errors, software has mass propagation of errors. The manager should prepare for maintenance during the development.

Studies of maintenance efforts have shown that most software defects after maintenance are caused by new features inducing defects in existing features, followed by defects in the new features themselves, defects remaining from before the maintenance effort, and defects induced by corrective changes. The most common type of defect is incorrect program logic, followed by missing logic, documentation defects, interface defects, and extra-logic defects. An awareness of these tendencies helps the manager plan for maintenance.

Managing maintenance involves reverse engineering, modifying existing items where safe, and making the task of doing maintenance clearly a valued one, so that people will not be turned off by an assignment to a maintainer's team.

25. Legal Issues I: Patentability and Copyright

Reading: Chisum86, Newell86

Software technology has given rise to a number of new legal questions. Should software be treated as goods or services under commercial law? What are the tax consequences? How should software be protected under intellectual property law: copyright, trade secret, patent, or a new form?

Intellectual property law provides a framework for allocating rights in innovative technology, provides incentives for investing resources, and may limit rights to reuse, rehost, retarget, translate, or otherwise modify software. The systems of intellectual property protection have developed independently but possess common elements, and they can profoundly affect the software development process at in all phases. An intellectual property system has a structure consisting of subject matter; a set of requisites; a set of exclusive rights; a set of limitations on exclusive rights; and infringement standards, process, and remedies. The copyright law assigns five rights to the creator of an intellectual property: the right to reproduce copies, prepare derivative works, distribute copies, perform publicly, and display publicly.

An item is not protected by an intellectual property system if it has been placed in the public domain (not the same as publishing). Such an item is usable by the public, but ineligible for further protection (except for improvements).

Intellectual property protection for software should create an incentive for innovative developments. It might have been based on patent law, copyright law, or sui generis (technology-specific) legislation. Congress chose copyright law as the basis. Courts now accept copyrightability of computer programs, including machine-readable versions. Some unanswered questions involve the right to reverse engineer software, to reuse, enhance, or modify software. Also unanswered are the implications of simultaneous copyright and trade secret protection.

In older cases software was never patentable, but courts have recently become more receptive to software patents. For example, certain types of algorithms have been held to be patentable. Data processing methodology has also been found to be patentable. To be patentable, a property must exhibit utility, novelty, and non-obviousness. Excluded from patentability are business systems, printed matter, and mental steps.

Chisum's definition of an algorithm: it is finite, definite, has specified inputs and outputs, and it is effective. An important current legal question is whether algorithms should be patentable.

26. Legal Issues II: Liability and Warranty

Reading: Holmes82, Friedman87, Cottrell86

What is the difference between "express" and "implied" warranty? How can we differentiate between goods and services? Forms of protection from liability include limited warranty, disclaimers, and limit of remedy. Fraud can be based on misrepresentation of a capability, using the user as a beta test site, misrepresentation of suitability or fitness, misrepresentation of time or management savings.

27. Special Cases

Reading: unpublished manuscript by the guest lecturer

[Guest lecturer: Harvey Hallman, Software Engineering Institute]

Special cases in project management include what to do when a project is failing and taking over a project already in progress.

There are three perspectives of a failing project: the team is failing, the team of teams is failing, the personal project is failing. We assume that the basic project management structures exist: a build process is in place, an error tracking system is in place, there is a design change request and review process, inspection processes are in place, configuration management is under control, and a test plan is in place.

The manager should look for symptoms of trouble ahead, determine why the project is failing, identify the reason for the failure, don't keep it a secret, and be prepared to reassign responsibilities. The manager should concentrate on the build process, find the area with the highest error rates, and selectively reduce the functionality of the product (saving full functionality for a later release).

Symptoms of trouble ahead include: the first missed schedule, confusing technical presentations, daily acceptance of requirements changes, a high level of design change requests, a high level of inspection and test problems, several reinspections of the same module, low morale, high employee overtime rate, build size growing faster than planned, and regression testing taking longer than planned.

Determining why the project is failing may uncover bad schedules, bad technology, or bad people. To identify the reason for failure, conduct an in-depth technical review and a detailed schedule review (to identify areas behind schedule and ahead of schedule).

Concentrating on the build process requires weekly meetings of representatives of each team for the purpose of reviewing the previous meeting's action items, the status of all activities in the current build path (code, error fixes, design changes, testing progress, and build bottlenecks), the progress on future activities. A result is to assign action items on each critical path item.

When your own personal project is failing: determine why is it failing; examine the possibility of overtime; work at home to gain perspective; set intermediate checkpoints for possible recovery of the schedule; identify functions to be delayed; let your boss know that you are not going to meet schedules.

When taking over a project already in progress, the most important step is to understand the project quickly: the architecture, terminology, the project plan and how well progress tracks to the plan, and what the project staff intend to do to get the job finished. The manager should determine if the project is in trouble, identify missing practices, determine the worry factor, identify the supporters and non-supporters, and search for depth of thought. The manager should act upon training and beliefs: don't compromise. Obtain buy-in from the staff, convert the non-supporters, and put the missing plans in place. Establish checkpoints, monitor the plans and checkpoints, shift people to reduce problem areas, and start recruiting immediately. Before taking over a project in mid-stream, understand why the job is available. The manager must believe in the project and believe that the team can be made successful. Be sure there is enough time to recover, not just enough time to take the blame for the failure.

28. Assessing the Organization's Ability to do the Process

Reading: Humphrey88

The software process is a major opportunity area for software organizations. The SEI has developed an approach to characterizing and assessing the software process. SEI software process assessments are an effective means for examining software processes and provide committed organizations a framework for action. The three leverage points are process, people, and technology; these are the determinants of software cost, schedule, and quality performance. The fundamental process management premise is: "The quality of a software system is governed by the quality of the process used to develop and evolve it."

The definition of a software process is: a set of activities, methods, and practices which guide people (with their software tools) in the production of software. The elements of process management are statistical control, process definition, process and product certification, and process support. The approach to the development of a better process is examine the characteristics of successful software groups; consider other technical fields; develop a hierarchy of software process management capability; produce an evaluation method; and apply and refine the method. An immature process is ad hoc and poorly controlled, highly dependent on current practitioners, and has unpredictable performance on cost, schedule, and quality. A mature process is defined and documented, well controlled, measured, focused on process improvement, and supported by technology. The benefits of a mature process are a more appropriate match between human skills and manual activities, increased likelihood of successful introduction of appropriate technology, and sustained orderly improvement of software production capabilities.

The SEI uses a five-level process maturity model; the levels are initial, repeatable, defined, managed, and optimizing. Each has recognizable characteristics and key problems; process improvement should be aimed at solving the problems at the current level. A result of the improvement process is a steady reduction of risk and steady growth of productivity and quality. Major changes in the process must start at the top of an organization; the process is a management responsibility.

Management must set challenging goals, provide the necessary resources, monitor the progress, and insist on performance. The goal is to fix the process, not the people: avoid the tendency to search for the guilty when something goes wrong and recognize that people, by nature, desire to do good work. A focus on people causes resistance to change.

An effective change program requires understanding of the current status. From that point, change is continuous: reactive changes generally make things worse, every defect is an improvement opportunity, and crisis prevention is more important than crisis recovery. Improvement requires investment: to improve the process, someone must work on it. Unplanned process improvement is wishful thinking. Improvements should be made in small steps. Training is expensive—but not nearly as expensive as not training.

The general improvement paradigm is: understand the current status, develop a vision of the desired state, establish a list of improvement actions in priority order, produce a plan to accomplish these actions, and commit the resources needed to execute the plan. Understanding the current status can begin with a process assessment. Its objectives are to understand an organization's current software engineering practices, identify key areas for process improvement, and facilitate the initiation of improvement actions by providing a framework for action and helping to obtain support for action. The benefits of an assessment include understanding an organization's software engineering capability with respect to the rest of the industry, preparing for potential contractor evaluations, providing an organization with a structured framework for periodic examination of status and needs, and helping to focus improvement efforts.

Assessment principles include involvement by senior management, a basis of process framework, confidentiality, collaboration, and action orientation. Some risks include morale problems if the assessment is not followed by action, inappropriate findings if the assessment is conducted as an audit, and failure of the action plan if the organization is not ready for change. The golden rule of assessment is, "Be prepared to take action or don't assess."

29. Final Examination

Bibliography

Abdel-Hamid86	Abdel-Hamid, Tarek K. and Madnick, Stuart E. "Impact of Schedule Estimation on Software Project Behavior." *IEEE Software 3*, 4 (July 1986), 70-75.
Albrecht83	Albrecht, Allan J. and Gaffney, John E., Jr. "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation." *IEEE Trans. Software Eng. SE-9*, 6 (Nov. 1983), 639-648.
Basili87	Basili, Victor R., Selby, Richard W., and Baker, F. Terry. "Cleanroom Software Development: An Empirical Evaluation." *IEEE Trans. Software Eng. SE-13*, 9 (Sept. 1987), 1027-1037.
Bernstein81	Bernstein, L. "Software Project Management Audits." *J. Syst. and Software 2*, 4 (Dec. 1981), 281-287.
Bersoff84	Bersoff, Edward H. "Elements of Software Configuration Management." *IEEE Trans. Software Eng. SE-10*, 1 (Jan. 1984), 79-87.
Boehm81	Boehm, Barry W. *Software Engineering Economics*. Englewood Cliffs, N. J.: Prentice-Hall, 1981.
Boehm84	Boehm, Barry W. "Software Engineering Economics." *IEEE Trans. Software Eng. SE-10*, 1 (Jan. 1984), 4-21.
Boehm87	Boehm, Barry W. "Improving Software Productivity." *Computer 20*, 9 (Sept. 1987), 43-57.
Boehm88	Boehm, Barry W. "A Spiral Model of Software Development and Enhancement." *Computer 21*, 5 (May 1988), 61-72.
Brooks75	Brooks, Frederick P., Jr. *The Mythical Man-Month: Essays on Software Engineering*. Reading, Mass.: Addison-Wesley, 1975. The book was "reprinted with corrections" in January 1982.
Buckley84	Buckley, Fletcher J. and Poston, Robert. "Software Quality Assurance." *IEEE Trans. Software Eng. SE-10*, 1 (Jan. 1984), 36-41.
Chisum86	Chisum, Donald S. "The Patentability of Algorithms." *U. of Pittsburgh Law Review 47*, 4 (Summer 1986), 959-1022.

Collofello87 Collofello, James S. and Buck, Jeffrey J. "Software Quality Assurance for Maintenance." *IEEE Software 4*, 9 (Sept. 1987), 46-51.

Cori85 Cori, Kent A. "Fundamentals of Master Scheduling for the Project Manager." *Project Management J. 16*, 2 (June 1985), 78-89.

Cottrell86 Cottrell, Paul and Maron, James. "Professional Liability for Computer Design." *The Computer Lawyer 3*, 8 (Aug. 1986), 14-20.

Crawford85 Crawford, Stewart G. and Fallah, M. Hosein. "Software Development Process Audits—A General Procedure." *Proc. 8th Intl. Conf. on Software Engineering*. Washington, D. C.: IEEE Computer Society Press, 1985, 137-141.

Cupello88 Cupello, James M. and Mishelevich, David J. "Managing Prototype Knowledge/Expert System Projects." *Comm. ACM 31*, 5 (May 1988), 534-541.

Dart87 Dart, Susan A., Ellison, Robert J., Feiler, Peter H., and Habermann, Nico. "Software Development Environments." *Computer 20*, 11 (Nov. 1987), 18-28.

DoD88 *Military Standard for Defense System Software Development*. DOD-STD-2167A, U. S. Department of Defense, Washington, D. C., Feb. 1988.

Fagan76 Fagan, Michael. "Design and Code Inspections to Reduce Errors in Program Development." *IBM Systems J. 15*, 3 (1976), 182-211.

Friedman87 Friedman, Marc S. and Hildebrand, Mary J. "Computer Litigation: A Buyer's Theories of Liability." *The Computer Lawyer 4*, 12 (Dec. 1987), 34-38.

Harvey86 Harvey, Katherine E. *Summary of the SEI Workshop on Software Configuration Management*. Tech. Rep. CMU/SEI-86-TR-5, ADA200085, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Dec. 1986.

Holmes82 Holmes, Robert A. "Application of Article Two of the Uniform Commercial Code to Computer System Acquisitions." *Rutgers Computer and Technology Law J. 9*, 1 (1982), 1-26.

Humphrey88 Humphrey, Watts S. "Characterizing the Software Process: A Maturity Framework." *IEEE Software 5*, 3 (Mar. 1988), 73-79.

Kemerer87 Kemerer, Chris F. "An Empirical Validation of Software Cost Estimation Models." *Comm. ACM 30*, 5 (May 1987), 416-429.

Metzger87 Metzger, Philip W. *Managing Programming People: A Personal View*. Englewood Cliffs, N. J.: Prentice-Hall, 1987.

Mills83	Mills, Harlan D. *Software Productivity*. Boston, Mass.: Little, Brown, 1983. Reprinted by Dorset House in 1988.
NASA86a	*Software Management Plan Data Item Description*. NASA-Sfw-DID-02-ADA, National Aeronautics and Space Administration, Office of Safety, Reliability, Maintainability, and Quality Assurance, Washington, D. C., 1986.
NASA86b	*Lessons-Learned Document Data Item Description*. NASA-Sfw-DID-41, National Aeronautics and Space Administration, Office of Safety, Reliability, Maintainability, and Quality Assurance, Washington, D. C., 1986.
Newell86	Newell, Allen. "Response: The Models Are Broken, the Models Are Broken!" *U. of Pittsburgh Law Review 47*, 4 (Summer 1986), 1023-1035.
Symons88	Symons, Charles R. "Function Point Analysis: Difficulties and Improvements." *IEEE Trans. Software Eng. SE-14*, 1 (Jan. 1988), 2-11.
Weinberg84	Weinberg, Gerald M. and Freedman, Daniel P. "Reviews, Walkthroughs, and Inspections." *IEEE Trans. Software Eng. SE-10*, 1 (Jan. 1984), 68-72.

4. Survey of Graduate Degree Programs in Software Engineering

Graduate degree programs first appeared in the late 1970s at Texas Christian University, Seattle University, and the Wang Institute of Graduate Studies. All three programs responded to needs of local industry in the Dallas/Fort Worth, Seattle, and Boston areas, respectively. In 1985, three additional programs were started: at the College of St. Thomas in St. Paul, Minnesota (now the University of St. Thomas), at Imperial College of Science and Technology in London, and at the University of Stirling in Scotland. The last five years have seen a significant increase in the development of and interest in such programs. We know of at least a dozen programs that either have been initiated or are under development.

In this section, we survey the programs for which we were able to obtain information. Readers will note substantial variation among the more established programs. This can be attributed to a number of factors:

- Most of the programs were developed in the absence of any recognized model curriculum.
- Each school had a number of existing courses, mostly in computer science, that were incorporated into the new programs; and these courses differed greatly among schools.
- Software engineering is a new discipline, and the developers of these programs had differing perceptions of the scope of the discipline, and its principles and practices.
- Each school was responding to perceived needs that varied from one community to another.

On the other hand, nine of the programs acknowledge being influenced by the SEI model curriculum, either directly or through the courses in the Academic Series. We hope that these schools' experiences will suggest improvements in the model curriculum.

Another notable point of variation among these programs is the program title (see Figure 4.1). Many of the programs were unable to use the word *engineering* in their titles because of legal or administrative restrictions. In one way, it is unfortunate that the term *software engineering* is so nearly universally accepted as an informal name for the discipline: an inordinate amount of time and energy has been devoted to arguing semantic issues of whether software engineering is really engineering, rather than to defining what constitutes an appropriate body of knowledge for professionals to learn.

We believe it is valuable for a school considering the development of a graduate program in software engineering to examine not only the SEI recommendations but also these existing programs. Therefore we have sketched the requirements for each program below.

Degree Title	**University**
Master of Software Engineering	Carnegie Mellon University Seattle University [Wang Institute of Graduate Studies]
Master of Science in Software Engineering	Andrews University Georgia Institute of Technology Monmouth College National University University of Houston-Clear Lake University of Pittsburgh University of Scranton University of Stirling
Master of Science in Software Systems Engineering	Boston University George Mason University
Master of Software Design and Development	Texas Christian University University of St. Thomas
Master of Science in Software Development and Management	Rochester Institute of Technology
Master of Computer Science, Software Engineering Option	Florida Atlantic University The Wichita State University
Master of Science in Computer Science, Software Engineering Option	University of West Florida
Master of Science (Computer Science)	Air Force Institute of Technology
Master of Science (Computer Systems)	Air Force Institute of Technology
Master of Systems Analysis	Miami University
Master of Science in Software Systems Management	Air Force Institute of Technology
Master of Science in Computational Systems Engineering, Software Engineering Option	Instituto Tecnológico y de Estudios Superiores de Monterrey
Master of Engineering	Imperial College of Science and Technology
Software Engineering Curriculum Master	Polytechnic University of Madrid

Figure 4.1. Titles of software engineering degree programs

Air Force Institute of Technology

Location	Wright Patterson Air Force Base, Ohio
Degree title	Master of Science (Computer Science) Master of Science (Computer Systems)
Degree requirements	Twelve required courses, one elective course in the theory area, and a thesis. The requirements are structured as six courses in systems, two in theory, two in an application sequence (see below), and one each in mathematics and technical communication.
Required courses	Systems and Software Analysis Software Design Software Generation and Maintenance Software Project Management Operating Systems Computer Architecture Principles of Embedded Software Formal-Based Methods in Software Engineering Advanced Information Structures Automata and Formal Language Theory Probability and Statistics for Computer Science Technical Reports and Thesis
Program initiation	See below.
Contact	Major Paul D. Bailor Department of Electrical and Computer Engineering Air Force Institute of Technology Wright Patterson Air Force Base, OH 45433
Source	This information was reported to the SEI by AFIT in August 1990.

The objective of the graduate programs in computer systems and computer engineering is the development of a broad competence in the application of the concepts and techniques of computer systems, computer science, and computer engineering, emphasizing specialized areas of interest to the Air Force. Each student is required to take a set of six systems courses (four of which are software engineering courses), a set of three theory courses, an application sequence, a graduate-level mathematics course, a technical writing and speaking course, and an independent study that leads to the preparation and completion of a master's thesis. Currently, seven application sequences are offered: software engineering, computer graphics, database systems, computer architecture, VLSI design, information systems, and artificial intelligence. The breadth of the systems and theory courses and the specialized application sequence courses prepare the students for a variety of Air Force assignments involving research, development, and program management in the career areas of computer systems, computer science, and computer engineering.

Courses in software engineering were introduced into the curriculum in the late 1970s. The application sequence in software engineering was developed in mid-1980s.

Air Force Institute of Technology

Location	Wright Patterson Air Force Base, Ohio
Degree title	Master of Science in Software Systems Management
Degree requirements	Seventeen required courses and a thesis. The requirements are structured as four technically-oriented software engineering courses, four management-oriented software engineering courses, one course in computer systems concepts, and eight courses in management and quantitative/qualitative analysis.
Required courses	Systems and Software Analysis Software Design Software Generation and Maintenance Principles of Embedded Systems Software Quality Assurance Software Cost and Schedule Estimation Software Configuration Management Seminar in Software Systems Management Computer Systems Concepts Managerial Economics Managerial Statistics I and II Theory and Practice of Professional Communications Introduction to Management Science Organization and Management Theory Organizational Behavior Federal Financial Management Contracting and Acquisition Management
Program initiation	June 1990
Contact	Major Chris Arnold Department of System Acquisition Management Air Force Institute of Technology Wright Patterson Air Force Base, OH 45433
Source	This information was reported to the SEI by AFIT in August 1990.

The objective of the graduate program in software systems management is to provide military and civilian software managers with the concepts, analytical skills, and methods of software systems management so that its graduates are prepared to handle the acquisition and management of large software systems, including embedded software systems. Each student is required to take a set of four technically-oriented software engineering courses, a set of management-based software engineering courses, a computer systems concepts course, additional courses in management and quantitative/qualitative analysis, and an independent study that leads to the preparation and completion of a master’s thesis.

Andrews University

Location	Berrien Springs, Michigan
Degree title	Master of Science in Software Engineering
Degree requirements	48 quarter credits (typically 4 credits per course): 8 credits of projects, 16 credits core courses, 0-20 credits foundation courses, 4-24 credits electives.
Foundation courses	Data Structures Data Base Systems Systems Analysis I Systems Analysis II Operating Systems
Core courses	Computer Architecture Software Engineering I Software Engineering II Programming Project Management
Program initiation	*[unknown]*
Contact	Daniel R. Bidwell Computer Information Science Dept. Andrews University Berrien Springs, MI 49104
Source	This information was reported to the SEI by Andrews University in April 1989.

Boston University

Location	Boston, Massachusetts
Degree title	Master of Science in Software Systems Engineering
Degree requirements	Nine courses of four credits each: seven required courses (including a project course) and two electives. Two of the required courses differ depending on whether the student's background is in hardware or software.
Required courses	Applications of Formal Methods Software Project Management Software System Design Computer as System Component Software Engineering Project Advanced Data Structures *(hardware background)* Operating Systems *(hardware background)* Switching Theory and Logic Design *(software background)* Computer Architecture *(software background)*
Program initiation	Fall 1988 (The program has existed as a software engineering option in the Master of Science in Systems Engineering since spring 1980; the current curriculum was adopted in January 1988.)
Contact	*[unknown]*
Source	This information was taken from [Brackett88].

Boston University absorbed the Wang Institute's facilities in 1987 and was the beneficiary of some of the experience of the Wang Institute. This program incorporates the best features of the MSE curriculum of Wang and the MS in Systems Engineering from Boston University. The program emphasizes the understanding of both hardware and software issues in the design and implementation of software systems. Special emphasis is placed on the software engineering of two important classes of computer systems: embedded systems and networked systems.

Both full-time and part-time programs are available, and most of the program is available through the Boston University Corporate Classroom interactive television system. The program can be completed in twelve months by full-time students.

The university also has a doctoral program leading to the PhD in Engineering, with research specialization in software engineering.

Carnegie Mellon University

Location	Pittsburgh, Pennsylvania
Degree title	Master of Software Engineering
Degree requirements	Fifteen courses: six required courses, three electives, a theory course, a business course, two software engineering seminars, and a four-semester master's project.
Required courses	Software Systems Engineering Formal Methods in Software Engineering Advanced System Design Principles Software Creation and Maintenance Software Analysis Software Project Management
Electives	Graduate courses in computer science and business
Prerequisite note	Prospective students must have at least two years of experience working in a sizable software project.
Program initiation	September 1989
Contact	Mark Ardis Software Engineering Institute Carnegie Mellon University Pittsburgh, Pennsylvania 15213
Source	This information was reported to the SEI by CMU in July 1990.

The objective of Carnegie Mellon University's MSE program is to produce a small number of highly skilled experts in software system development. It is designed to elevate the expertise of practicing professional software designers. The emphasis is on practical application of technical results from computer science; the nature of these technical results dictates a rigorous, often formal, orientation. The engineering setting requires responsiveness to the needs of end users in a variety of application settings, so the program covers resolution of conflicting requirements, careful analysis of tradeoffs, and evaluation of the resulting products. Since most software is now produced by teams in a competitive setting, the program also addresses project organization, scheduling and estimation, and the legal and economic issues of software products.

Florida Atlantic University

Location	Boca Raton, Florida
Degree title	Master of Computer Science, Software Engineering Option
Degree requirements	33 semester hours, including three regular FAU courses, five of the six FAU/SEI videotape courses, and CASE tools material (may or may not be a separate course).
Required FAU courses	Compiler Writing Computability and Complexity Artificial Intelligence
FAU/SEI videotape courses	Software Project Management Software Systems Engineering Specification of Software Systems Principles and Applications of Software Design Software Generation and Maintenance Software Verification and Validation
Admissions note	The software engineering option is available only to students participating in the FAU/SEI videotape courses offered in cooperation with specific south Florida companies.
Program initiation	September 1989
Contact	Neal Coulter Department of Computer Science Florida Atlantic University PO Box 3091 Boca Raton, FL 33431-0991
Source	This information was reported to the SEI by Florida Atlantic University in December 1989.

George Mason University

Location	Fairfax, Virginia
Degree title	Master of Science in Software Systems Engineering
Degree requirements	30 hours of course work in the School of Information Technology and Engineering, including six required courses.
Required courses	Software Construction Software Requirements and Prototyping Software Design Formal Methods and Models in Software Engineering Software Project Management Software Project Laboratory
Electives	Four courses, or two courses and 6 semester hours of master's thesis.
Program initiation	Fall 1989 (core courses offered beginning Fall 1988)
Contact	Hassan Gomaa School of Information Technology George Mason University 4400 University Drive Fairfax, VA 22030
Source	This information was reported to the SEI by George Mason University in August 1990.

The program for the degree of Master of Science in Software Systems Engineering is concerned with engineering technology for developing and modifying software components in systems that incorporate digital computers. The program is concerned with both technical and managerial issues, but primary emphasis is placed on the technical aspects of building and modifying software systems.

In addition to the degree program, the university offers a Graduate Certificate Program in software systems engineering. The program is designed to provide knowledge, tools, and techniques to those who are working in, or plan to work in, the field of software systems engineering, but do not desire to complete all of the requirements for a master's degree. Students in the certificate program must already hold or be pursuing a master's degree in a science or engineering discipline. To receive the certificate students must complete the six required courses listed above.

Georgia Institute of Technology

Location	Atlanta, Georgia
Degree title	Master of Science in Software Engineering
Degree requirements	50 quarter hours of coursework, including nine required courses, four electives, and a three-quarter software engineering project sequence.
Required courses	Introduction to Software Engineering Foundations of Software Engineering Programming Language Design Human Computer Interface Requirements Analysis and Prototyping Specification of Software Systems Project Management Principles and Applications of Software Design Software Generation, Test, and Maintenance Software Engineering Project I, II, III
Admissions note	Entering students must have an appropriate undergraduate degree (typically in computer science) and at least two years of full-time software development experience.
Program initiation	This program has been proposed; it has not yet been approved.
Contact	Not yet designated
Source	This information was reported to the SEI by the Georgia Institute of Technology in November 1990.

Georgia Tech has recently created a College of Computing in recognition of the importance of the computing-related disciplines. The college recognizes the need within the computer industry for professionals able to provide technical and managerial leadership in the area of software engineering.

The curriculum most appropriate to the traditions and capabilities of the Institute and of the College of Computing falls between the extremes of very theoretical and completely practical. The program should emphasize practical skills that will equip graduates to play leadership roles in the software industry. At the same time, they should develop a sufficient fundamental understanding of software engineering to enable them to keep up with changes in a rapidly growing and evolving field. The best way to characterize this dual emphasis is to say that the curriculum leads to a “professional” degree.

Imperial College of Science and Technology

Location	London, England
Degree title	Master of Engineering
University structure	British universities normally have three-year bachelor's degree programs; the master of engineering is a four-year first degree program. In its first two years the program is the same as the (three-year) bachelor of science program in computer science.
Degree requirements	Third and fourth year coursework includes compulsory courses totaling three modules and optional courses totaling six modules (each module represents 22 hours of lecture). During the third year, students spend approximately six months in industry; during the fourth year they must complete an individual project.
Compulsory courses	*(these courses total six modules)* Software Engineering Process Calculus of Software Development Database Technology Introduction to Macro Economics and Financial Management Introduction to Management Methodology of Software Development Language Definition and Design Programming Support Environments Standards, Ethical and Legal Considerations
Optional courses (third year)	*(one module each)* Functional Programming Technology I Artificial Intelligence Technology Compiler Technology Computer Networks Object Oriented Architecture Interface and Microprocessor Technology Performance Analysis of Computer Systems Graphics Silicon Compilation Applied Mathematics Industrial Sociology Government Law and Industry Humanities

Optional courses (fourth year)	*(one module each)* Advanced Logic Theorem Proving Concurrent Computation Human-Computer Interaction Expert Systems Technology Functional Programming Technology II Advanced Operation Systems Parallel Architecture Distributed Systems VLSI Robotics Computing in Engineering Natural Language Processing Micro-Economic Concepts Industrial Relations Innovation and Technical Change Humanities
Program initiation	Fall 1985
Contact	*[unknown]*
Source	This information was taken from [Lehman86].

Since British students normally must commit to either a three-year (bachelor's degree) or a four-year (master's degree) program at the end of secondary school (the student cannot complete the bachelor's degree and then decide to continue for the master's), the latter programs tend to attract the better students. Entrance requirements are generally more stringent for the master's programs, and the graduates are expected to advance rapidly once they enter industry.

The industry component of this program has been described earlier in this report (Section 2.7). This component is perceived to be somewhat analogous to the role of teaching hospitals in the education of medical students.

Instituto Tecnológico y de Estudios Superiores de Monterrey
(Technological and Higher Learning Institute of Monterrey)

Location	Monterrey, Nuevo León, Mexico
Degree title	Master of Science in Computational Systems Engineering. Software Engineering Option
Degree requirements	Fourteen courses totalling 150 units: six core courses (72 units), three elective courses (36 units), two personal development workshops (12 units), research methods seminar (6 units), and thesis (24 units).
Core courses	Discrete Mathematics Analysis of Algorithms Theory of Computation Software Design and Development Software Analysis, Design and Specification Software Generation, Verification, and Maintenance
Program initiation	1990
Contact	Prof. Carlos Scheel Mayenberger Programa de Graduados en Informatica Instituto Tecnologico y de Estudios Superiores de Monterrey Sucursal de Correos "J" C.P. 64849 Monterrey, N. L. México
Source	This information was reported to the SEI by ITESM in October 1990.

The software engineering option of the program is structured for completion in four semesters, or three semesters and two summers. The program also has options in distributed systems and artificial intelligence.

Miami University

Location	Oxford, Ohio
Degree title	Master of Systems Analysis
Degree requirements	30 semester hours: twelve hours of core courses, twelve hours of systems electives, and six hours of graduate research.
Core courses	Analysis of Information Systems *plus any three of:* Structured Design and Implementation Data Structures and Data Base Systems Operations Research II Simulation Artificial Intelligence
Systems electives	Advanced Software Engineering Advanced Data Base Systems Data Communication Networks & Distributed Process Expert Systems Operating Systems Concepts Advanced Simulation Analysis of Inventory Systems Analysis of Forecasting Systems Analysis of Manufacturing Systems Regression Analysis An Introduction to Applied Probability Seminar in Systems Analysis
Prerequisite note	Students with little formal education or experience in systems analysis or related disciplines may be required to complete up to 13 semester hours of additional foundation courses.
Program initiation	Fall 1990
Contact	Mufit Ozden Department of Systems Analysis Miami University Oxford, Ohio 45056
Source	This information was reported to the SEI by Miami University in January 1990.

The aim of the program is to graduate a systems analyst who has a sound grasp of systems development and the mathematical models frequently needed in industrial information systems. It differs from computer science programs through its strong focus on the practical aspects of systems development and mathematical models. It differs from MIS programs offered by schools of business through its technical emphasis on systems development built on a solid foundation of computer science and mathematics.

Monmouth College

Location	West Long Branch, New Jersey
Degree title	Master of Science in Software Engineering
Degree requirements	30 credit hours, consisting of six core and four elective courses.
Core courses	Mathematical Foundations of Software Engineering I Software Engineering Project Management Formal Methods in Programming Software Systems Design System Project Implementation (Laboratory Practicum)
Elective courses	Mathematical Foundations of Computer Science II Computer Communications Programming Languages Database Systems Security Aspects of Systems Design System Development Environment Technology AI Technology for Software Engineers Software Quality
Program initiation	1986
Contact	Richard Kuntz Monmouth College West Long Branch, New Jersey 07764
Source	This information was taken from [Amoroso88] and from information reported to the SEI by Monmouth College in July 1990.

The program is offered through the departments of computer science and electrical engineering. The current enrollment is more than 100, and to date 50 students have completed the degree requirements.

National University

Location	San Diego, California
Degree title	Master of Science in Software Engineering
Degree requirements	60 quarter units, of which at least 45 units (including the software engineering project courses) must be completed in residence.
Required courses	Principles of Software Engineering Advanced Software Engineering Verification and Validation Techniques Principles of Hardware and Software Integration Systems Software Networked Computing Systems Data Base Management I, II Expert Systems Software Engineering Project I, II, III
Prerequisite note	Programming ability in Ada is a prerequisite.
Program initiation	April 1985
Contact	*[unknown]*
Source	This information was reported to the SEI by National University in December 1989.

National University is the third largest private university in California, with more than 10,000 students currently enrolled. It has over 100 students in the MSSE program at campuses in San Diego, San Jose, Sacramento, Irvine, Los Angeles, and Vista. As of December 1989, more than 400 students have graduated from the MSSE program.

Graduate classes meet for 40 hours over a four week period, primarily in the evening in order to accommodate the schedules of working adults. Approximately 85% of the students in the MSSE program are currently software practitioners.

Most instructors in the program are adjunct faculty who work for local companies and who are recognized experts in their fields.

Polytechnic University of Madrid

Location	Madrid, Spain
Degree title	Software Engineering Curriculum Master
University structure	The Spanish university system organizes its programs differently from United States universities, so this program cannot be described in terms of courses. For each of the subject areas described below, the amount of time devoted to the area is given in units. Each unit represents a 75-minute class meeting. The program totals approximately 500 units.
Degree requirements	Introduction to Software Engineering (3) Models of Computation (76) Computing Machinery (6) Software Production Technology and Methodology Information Systems Introduction to Requirements Analysis (15) Formal Specification Techniques (25) Design (55) Implementation (85) Tools Evaluation (2) Software Engineering and Artificial Intelligence (11) Product and Process Control System Construction Management (20) Quality Control Project Management (20) Documentation Process (25) Software Product (8) Information Protection (14) Software Safety (8) Legal Aspects (6) Case Study (12)
Program initiation	1988
Contact	*[unknown]*
Source	This information was reported to the SEI by Polytechnic University in May 1989.

The Polytechnic University of Madrid is the largest (well over 100,000 students) and most prestigious of the Spanish technical universities. It has large, well-established schools of engineering and informatics (computer science). The university is an academic affiliate of the SEI and has incorporated a number of SEI recommendations into its initial curriculum.

Rochester Institute of Technology

Location	Rochester, New York
Degree title	Master of Science in Software Development and Management
Degree requirements	48 credits (quarter system; typical course is four credits).
Required courses	Principles of Software Design Principles of Distributed Systems Principles of Data Management Software and System Engineering Project Management Organizational Behavior Analysis and Design Techniques, *or* Analysis & Design of Embedded Systems Software Verification and Validation Software Project Management Technology Management Software Tools Laboratory Software Engineering Project
Program initiation	Fall 1987
Contact	Jeffrey A. Lasky Graduate Department of Computer Science Rochester Institute of Technology PO Box 9887 Rochester, NY 14623-0887
Source	This information was reported to the SEI by RIT in April 1989.

The program has approximately 100 students at the RIT campus and 15 students at Griffiss Air Force Base in Rome, New York. Approximately 90% of the students attend part-time.

Seattle University

Location	Seattle, Washington
Degree title	Master of Software Engineering
Degree requirements	45 credits (quarter system), including eight required core courses, four elective courses, and a project sequence extending over three quarters.
Required courses	Technical Communication Software Systems Analysis System Design Methodology Programming Methodology Software Quality Assurance Software Metrics Software Project Management Formal Methods
Elective courses	System Procurement Contract Acquisition and Administration Database Systems Distributed Computing Artificial Intelligence Human Factors in Computing Data Security and Privacy Computer Graphics Real Time Systems Organization Behavior Organization Structure and Theory Decision Theory *(other electives may be selected from the MBA program)*
Prerequisite note	Prospective students must have at least two years of professional software experience.
Program initiation	1979
Contact	Everald E. Mills Software Engineering Department Seattle University 900 Broadway Avenue Seattle, WA 98122
Source	This information was taken from [Mills86], with additional information reported to the SEI by Seattle University in July 1990.

Seattle University is an independent urban university committed to the concept of providing rigorous professional educational programs within a sound liberal arts background. In 1977 the university initiated a series of discussions with representatives from local business and industry, during which software engineering emerged as a critical area of need for specialized educational programs. Leading software professionals were invited to assist in the development of such a program, which was initiated the following year.

Normally, classes are held in the evenings and students are employed full-time in addition to their studies. The first graduates of the program received MSE degrees in 1982.

Texas Christian University

Location	Fort Worth, Texas
Degree title	Master of Software Design and Development
Degree requirements	36 semester hours, including nine required courses and three electives; submission of a technical paper to a journal for publication.
Required courses	Introduction to Software Design and Development Modern Software Requirements and Design Techniques Applied Design, Programming, and Testing Techniques Management of Software Development Economics of Software Development Computer Systems Architecture Database and Information Management Systems Software Implementation Project I Software Implementation Project II
Program initiation	Fall 1978
Contact	James R. Comer Computer Science Department Texas Christian University Ft. Worth, TX 76129
Source	This information was taken from [Comer86] and reconfirmed by Texas Christian in July 1990.

The university established a graduate degree program in software engineering in 1978. Due to external pressure, prompted by the absence of an engineering college at TCU, the program was given its current name in 1980.

The program offers most of its courses in the evening, and all 50 students in the program are employed full-time in the Dallas/Fort Worth area.

University of Houston-Clear Lake

Location	Houston, Texas
Degree title	Master of Science in Software Engineering
Degree requirements	36 credit hours, including 30 hours of required courses and 6 hours of electives.
Required courses	Specification of Software Systems Principles and Applications of Software Design Software Generation and Maintenance Software Validation and Verification Software Project Management Master's Thesis Research Advanced Operating Systems Theory of Information and Coding Synthesis of Computer Networks
Elective courses	Must be chosen from courses in software engineering, computer science, computer systems design, or mathematical sciences.
Program initiation	September 1990
Contact	Dean E. T. Dickerson Office of the Dean University of Houston-Clear Lake Houston, TX 77058-1057
Source	This information was reported to the SEI by the University of Houston-Clear Lake in July 1990.

Five of the required courses in this degree program are based on SEI recommendations.

University of Pittsburgh

Location	Pittsburgh, Pennsylvania
Degree title	Master of Science in Software Engineering
Degree requirements	33 credits: four required software engineering courses; additional required and optional courses in computer science.
Required courses	Software Engineering: Specification and Design Software Engineering: Implementation and Testing Information Processing Systems Master's Directed Project *Either of:* Theory of Computation I Design and Analysis of Algorithms I *Any two of:* Language Design Advanced Computer Operating Systems I Computer Architecture
Elective courses	*Three graduate-level courses including two of:* Modeling and Simulation Principles of Database Systems Interface Design and Evaluation Knowledge Representation
Program initiation	1989
Contact	*[unknown]*
Source	This information was reported to the SEI by the University of Pittsburgh in the fall of 1990.

This program is project-oriented, emphasizes a methodological approach to software development, and provides a more focused education than the traditional Master of Science in Computer Science. Applicants with professional experience may be given special consideration for admission, although such experience is not required. All students' programs are individually designed with the help of a faculty advisor. There is no thesis requirement.

University of Scranton

Location	Scranton, Pennsylvania
Degree title	Master of Science in Software Engineering
Degree requirements	36 graduate credits: six required courses and four electives (3 credits each), and a thesis (6 credits)
Required courses	Introduction to Software Engineering Advanced Data Structures and Algorithms Formal Methods and Models Requirements Analysis and Software Specification Principles and Applications of Software Design Software Project Management
Electives	Software Generation and Maintenance Engineering of Software Systems Database Systems Cost Collection and Analysis Metrics Real-time and Embedded Systems CASE Tools Legal Aspects and Ethics
Program initiation	Fall 1990
Contact	Dr. J. Fernando Naveda Director, Master of Science in Software Engineering Department of the Computing Sciences University of Scranton Scranton, PA 18510-4664
Source	This information was reported to the SEI by the University of Scranton in August 1990.

The program expects 15 part-time students during the first year, with full-time students beginning in the second year.

The student body is expected to be composed of software practitioners, most of whom will not have a recent computer science degree or a strong background in some of the more formal methods of computer science. With this in mind, the program begins with two bridge courses, *Introduction to Software Engineering* and *Advanced Data Structures and Algorithms*. The goals of these courses are to give the students the mathematics needed in subsequent courses, an overview of what software engineering is (the “big picture”), and knowledge of data structures in Ada.

The university does not offer a graduate degree in computer science.

University of Stirling

Location	Stirling, Scotland
Degree title	Master of Science in Software Engineering
Degree requirements	Semester 1 (September-December) Programming Methods Language Concepts Introduction to Software Engineering Computing Science Structures and Techniques Initial industrial placement visits (January) Semester 2 (February-July) Methods for Formal Specification Concurrency (half semester) Databases (half semester) Networks and Communications Elective: Expert Systems or Language Implementation Industrial project (July-December) Dissertation (January-March)
Program initiation	1985
Contact	*[unknown]*
Source	This information was reported to the SEI by the University of Stirling in April 1989.

The MSc in Software Engineering is a "specialist conversion course" intended to train graduates with a scientific background in the methods of software engineering. The students spend twelve months at the University of Stirling and six months at an industrial research and development center. Through this approach students are given an understanding of both the current engineering technology and its application in an industrial context.

The six-month placement in industry enables each candidate to participate in a project and be responsible for a particular investigation. Where practical, this may form the basis of the individual project that is undertaken during a final three-month period and then written up in the dissertation.

University of St. Thomas

Location	St. Paul, Minnesota
Degree title	Master of Software Design and Development
Degree requirements	Ten required courses, including a two-semester project course sequence, and four elective courses. All courses are three semester credits.
Required courses	Technical Communications Software Engineering Methodologies DBMS and Design Systems Analysis and Design I Software Productivity Tools I Software Project Management Software Quality Assurance/Quality Control Legal Issues in Technology
Program initiation	February 1985
Contact	Bernice M. Folz, Dean Department of Quantitative Methods and Computer Science University of St. Thomas 2115 Summit Avenue St. Paul, MN 55105-1096
Source	This information was reported to the SEI by the University of St. Thomas in July 1990.

This program was developed through an advisory committee made up of technical managers from Twin Cities companies such as Honeywell, IBM, Sperry, 3M, NCR-Comten, and Control Data. Elective courses are added to the curriculum on the basis of need as expressed by technical managers in local industry or by students in the program.

The program is applied rather than research-oriented. Most instructors are from industry (14 of 23 in the spring 1990 semester). Instead of a thesis, students complete a two-semester software project in a local company; in many cases this company is their employer, but the project must not be part of their normal work responsibilities.

Classes are offered evenings, and 98% of students work full-time in addition to their studies. Students normally require three years to complete the degree. The program enrolled 290 students in spring 1990.

Prior to September 1, 1990, the school's name was the College of St. Thomas.

University of West Florida

Location	Pensacola, Florida
Degree title	Master of Science in Computer Science, Software Engineering Option
Degree requirements	33 semester hours of approved course work; at least 18 hours at 6000 (advanced) level; up to six hours of related course work; thesis optional.
Required courses	Advanced Operations Research Software Engineering Project Software Engineering Economics Software Engineering Management Computer Aided Software Engineering Computer Systems Performance Analysis Embedded Programming in Ada Advanced Database Systems
Prerequisites	In addition to the expected undergraduate computer science prerequisites, the program requires a two-semester sequence in software engineering, two semesters of economics, and one each of technical writing, management, operations research, and statistics.
Program initiation	1989
Contact	Theodore F. Elbert Professor and Division Head Division of Computer Science University of West Florida 11000 University Parkway Pensacola, Florida 32514-2542
Source	This information was reported to the SEI by the University of West Florida in July 1990.

The University offers three substantially different options within its Master of Science in Computer Science program, the other two being the Systems and Control Engineering option and an option simply referred to as the MSCS. The Software Engineering option provides instruction in advanced concepts of software engineering, database methodologies, and computer performance analysis. The Systems and Control Engineering option provides advanced course work in mathematics, modern control theory concepts, computer architecture, and software engineering as it applies to real-time embedded systems. The MSCS option provides advanced instruction in concepts of computer science, with concentration in the areas of artificial intelligence, knowledge-based systems, data classification, and image processing.

The requirements for the Software Engineering option will be revised during the 1990-91 academic year.

Wang Institute of Graduate Studies

Location	Tyngsboro, Massachusetts
Degree title	Master of Software Engineering
Degree requirements	Eleven courses: eight required courses (including two project courses) and three electives.
Required courses	Formal Methods Programming Methods Management Concepts Computing Systems Architecture *or* Operating Systems Software Project Management Software Engineering Methods Project I, II
Elective courses	Database Management Systems User Interface Design, Implementation and Evaluation Survey of Programming Languages Expert System Technology Translator Implementation Computing Systems Architecture Operating Systems Principles of Computer Networks Programming Environments
Prerequisite notes	Admission requirements included at least one year of full-time software development work experience. Also required was submission of a three to four page essay on a software development or maintenance project in which the applicant had participated, an expository survey of a technical subject, or a report on a particular software tool or method.
Program initiation	1979
Contact	None
Source	This information was taken from [Wang86].

The Wang Institute of Graduate Studies closed in the summer of 1987. Its facilities were donated to Boston University, and its last few students were permitted to complete their degrees at BU. During its existence, the Wang program was generally considered to be the premier program of its kind. Schools considering development of an MSE program would be well advised to examine the Wang program as a model.

Wang Institute was also a pioneer in the development of a very high quality faculty with renewable fixed-term contracts rather than a tenure system. For a rapidly evolving discipline such as software engineering, where the faculty's professional experience may be at least as valuable as its academic credentials, this model for faculty evaluation and retention may be worthy of consideration by other schools as well.

The Wichita State University

Location	Wichita, Kansas
Degree title	Master of Computer Science, Software Engineering Option
Degree requirements	30 credit hours total: two required courses, six credit hours of software engineering electives, additional electives in software engineering or computer science, and practicum (3 hours) or thesis (6 hours) on a software engineering topic.
Required courses	Software Requirements, Specification and Design Software Testing and Validation
Elective courses	Software Project Management Ada and Software Engineering Systems Analysis Topics in Software Engineering *(recent offerings have included Configuration Management, Formal Methods, Quality Assurance, Software Metrics, and Formal Verification of Software)*
Program initiation	Fall 1988
Contact	Mary Edgington, Chair Computer Science Department The Wichita State University Wichita, Kansas 67208
Source	This information was reported to the SEI by Wichita State in July 1990.

The Wichita State University Department of Computer Science has created a set of courses than can lead to a specialization in software engineering within the existing Master of Computer Science degree program. These courses are taught in cooperation with the Software Engineering Institute's Software Engineering Curriculum Project.

5. Survey of Comprehensive Software Engineering Textbooks

The growth of software engineering education has stimulated the writing and publication of many software engineering textbooks over the last few years. In response to many requests, the SEI Software Engineering Curriculum Project has begun developing an annotated bibliography of these books. The bibliography includes not only the usual information, but also the comments of professors who have used the books in their courses. From time to time we will publish the bibliography, or selected subsets of it, to aid other educators who wish to select appropriate texts for their courses.

The bibliography that follows contains comprehensive software engineering textbooks, meaning those that present the broadest view of all aspects of software engineering. These books are suitable for a one- or two-semester course in software engineering. More specialized books, such as those covering only requirements analysis or only testing, will be presented in future reports.

Software engineering educators are invited to suggest other books to be included in future bibliographies, and to submit comments or reviews of textbooks. These may be addressed to the Software Engineering Curriculum Project at the SEI; electronic mail may be sent to education@sei.cmu.edu on the Internet.

Russell J. Abbott. ***An Integrated Approach to Software Development***

New York: John Wiley & Sons Inc., 1986. ISBN 0-471-82646-4. 334 pages. $36.95. Includes exercises.

Table of Contents

Doug Bell, Ian Morrey, and John Pugh. ***Software Engineering—A Programming Approach***

Englewood Cliffs, N. J.: Prentice-Hall International, 1987. 250 pages.

Table of Contents

Frederick P. Brooks, Jr. ***The Mythical Man-Month: Essays on Software Engineering***

Reading, Mass.: Addison-Wesley, 1975. ISBN 0-201-00650-2. 195 pages. $20.50.

Table of Contents

Comment from the publisher

An eminent computer expert, Brooks has written a collection of thought-provoking essays on the management of computer programming projects. These essays draw from his own experience as project manager for the IBM System/360 and for OS/360, its operating system.

In the essays, the author blends facts on software engineering with his own personal opinions and the opinions of others involved in building complex computer systems. He not only gives the reader the benefit of the lessons he has learned from the OS/360 experience, but he writes about them in an extremely readable and entertaining way.

Although formulated as separate essays, the book expresses a central argument. Brooks believes that large programming projects suffer management problems different in kind from small ones due to the division of labor. For this reason he feels that the critical need is for conceptual integrity of the product itself, and in essay form he explores both the difficulties of achieving this unity and the methods for achieving it.

Comment by Professor Jim Tomayko, The Wichita State University

This is the single most useful text for sparking discussion. It should be used as a supplement in conjunction with more comprehensive texts, and cannot stand alone in most courses.

Richard Fairley. *Software Engineering Concepts*

New York: McGraw-Hill, 1985. ISBN 0-07-019902-7. 364 pages. $49.95. Includes exercises and suggested project ideas.

Reviewed in *Computing Reviews*, June 1987 and February 1989; *Software Engineering Journal*, July 1989.

Table of Contents

1. Introduction to Software Engineering
2. Planning a Software Project
3. Software Cost Estimation
4. Software Requirements Definition
5. Software Design
6. Implementation Issues
7. Modern Programming Language Features
8. Verification and Validation Techniques
9. Software Maintenance
10. Summary

Review by Professor Richard Hamlet, Portland State University

This book on general software engineering makes an adequate text for an undergraduate or survey course. Its primary weakness is a lack of depth, more important at the graduate level. However, since its organization is close to comprehensive, it can be supplemented with current papers to provide detail. It is also stronger on common-sense, management material than on engineering topics. For example, its treatment of abstract data types and information hiding is so superficial as to be useless, yet this should be a primary design topic in a technically oriented course. I do not believe that there is a good, technical software engineering text available; some would say that is because the subject has no technical content, and that books like Fairley's are the best that can be done. It is a weakness of the book that the references are not collected together in one place, but rather scattered at chapter endings: even though the paper you want is cited, you have to guess where. The exercises, although better than the traditional ones created at the last minute when the publisher suggests them, are too much of the flavor of "summarize Section __ in your own words," or "design something–how did it go?"

Comment by Professor Jim Tomayko, The Wichita State University

The actual textual material in this book is terse and minimalistic. Its strengths lie in its excellent references and in providing templates for the most commonly-developed documents.

Joseph M. Fox. *Software and its Development*

Englewood Cliffs, N. J.: Prentice-Hall, 1982. ISBN 0-13-822098-0. 299 pages. $46.00.

Table of Contents

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*

Englewood Cliffs, N. J.: Prentice-Hall, 1991. 573 pages.

Table of Contents

Philip Gilbert. *Software Design and Development*

Chicago: Science Research Associates, 1983. ISBN 0-574-21430-5. 681 pages. Includes systems, design, and evaluation problems. Instructor's guide is available.

Reviewed in *Computing Reviews*, February 1987.

Table of Contents

Watts S. Humphrey. *Managing the Software Process*

Reading, Mass.: Addison-Wesley, 1989. ISBN 0-201-18095-2. 494 pages. $43.25.

Reviewed in *American Programmer*, January 1990; that is review condensed and reprinted in *Data Processing Digest*, May 1990.

Table of Contents

Comment by Professor Jim Tomayko, The Wichita State University

> This is a good overview of the software development process, but it should be graduate-level only and should be used in conjunction with a more comprehensive text that explains specific methods.

Randall W. Jensen and Charles C. Tonies, eds. ***Software Engineering***

Englewood Cliffs, N. J.: Prentice-Hall, 1979. ISBN 0-13-822130-8. 580 pages. $56.00.

Table of Contents

Comment by Professor Jim Tomayko, The Wichita State University

Despite its advanced age, there is little in this book that is dated, with the exception of some of the material on legal issues. The appendix dealing with software engineering education still rings true. The book's strength is that it is written by industrial experts, and thus has little "academic" flavor. The weakness is that, even though the concepts remain the same, lots of the terminology has changed.

David Alex Lamb. ***Software Engineering: Planning for Change***

Englewood Cliffs, N. J.: Prentice-Hall, 1988. ISBN 0-13-822982-1. 298 pages. $47.00. Includes exercises.

Reviewed in *Computing Reviews*, February 1989; *IEEE Software*, November 1989; *Computer*, March 1988.

Table of Contents

A. Macro and J. N. Buxton. *The Craft of Software Engineering*

Wokingham, England: Addison-Wesley, 1987. ISBN 0-201-18488-5. 380 pages. $26.95.

Reviewed in *Software Engineering Journal*, July 1989; *Information and Software Technology*, December 1989.

Table of Contents

Allen Macro. *Software Engineering: Concepts and Definitions*

Prentice-Hall International, 1990. 544 pages. $43.20.

Table of Contents

Philip W. Metzger. ***Managing a Programming Project, 2nd Ed.***

Englewood Cliffs, N. J.: Prentice-Hall, 1981. ISBN 0-13-550772-3. 244 pages. $50.00.

Table of Contents

Comment by Gary Ford, SEI

As the title implies, this book focuses on project management. An especially useful part of the book is the detailed outline of a project plan, which runs 31 pages. The book can be a valuable resource to an instructor of a software engineering project course, even when the students are using another book.

Barbee Teasley Mynatt. ***Software Engineering with Student Project Guidance***

Englewood Cliffs, N. J.: Prentice-Hall, 1990. 429 pages.

Table of Contents

Shari Lawrence Pfleeger. ***Software Engineering: The Production of Quality Software***

New York: Macmillan, 1987. ISBN 0-02-395720-4. 443 pages. Includes exercises. Instructor's manual available; includes answers to exercises.

Reviewed in *Computing Reviews*, July 1988 and February 1989; *Software Engineering Journal*, July 1989.

Table of Contents

Roger S. Pressman. ***Software Engineering: A Practitioner's Approach, 2nd Ed.***

New York: McGraw-Hill, 1987. ISBN 0-07-050783-X. 567 pages. $44.95.

Reviewed in *IEEE Software*, January 1988.

Table of Contents

Comment by Professor Jim Tomayko, The Wichita State University

This book is a significant improvement on the first edition. It contains material at sufficient depth for a one-semester course. It goes beyond Fairley in that you truly feel as though you understand the usefulness and characteristics of a method or concept.

Ronald A. Radice and Richard W. Phillips. ***Software Engineering: An Industrial Approach, Vol. 1***

Englewood Cliffs, N. J.: Prentice-Hall, 1988. 457 pages. $37.00.

Reviewed in *IEEE Software*, September 1989.

Table of Contents

B. Ratcliff. ***Software Engineering—Principles and Methods***

Blackwell, 1987.

Reviewed in *Software Engineering Journal*, July 1989.

Kenneth D. Shere. ***Software Engineering and Management***

Englewood Cliffs, N. J.: Prentice Hall, 1988. ISBN 0-13-822081-6.

Table of Contents

Martin L. Shooman. *Software Engineering: Design, Reliability, and Management*

New York: McGraw-Hill, 1983. ISBN 0-07-057021-3. 683 pages. $50.95. Appendices summarize probability theory, reliability theory, and graph theory. Includes exercises and answers to selected exercises.

Table of Contents

Review by Professor J. H. Poore, The University of Tennessee

This is an excellent book but is a bit out of date. I usually make a course project of updating all the charts and tables from current reports. The approach of this book is sound. The organization of the material is correct and the methodology is solid. By using this book, students understand that the field is inherently mathematical and that the word *engineering* is not an affectation. I supplement the course with technical papers to update the models and introduce new ones. This book is used in a senior-level course.

Charles D. Sigwart, Gretchen L. Van Meer, and John C. Hansen. *Software Engineering: A Project Oriented Approach*

Irvine, Calif.: Franklin, Beedle & Associates, 1990. ISBN 0-938661-27-2. 484 pages.

Table of Contents

Review by Professor W. S. Curran, Southeastern Louisiana University

If you are looking for a good textbook for an introductory course on software engineering, Sigwart, Van Meer and Hanson's new text should please you greatly. It is a book written by experienced teachers of software engineering who are sympathetic to the problems inherent in group projects, and who are interested in producing students who are well prepared for the "real world." Too often, students have been burdened with arcane and archaic formulae, idiosyncratic and bewildering terminology and diagrams, or highly specialized topics in software engineering texts. In this book, the authors have concentrated instead on how to do software engineering properly by simulating an industrial environment and incorporating academic features as well. They use industry and IEEE standard terminology and diagrams throughout (with glossary), and have included a suggested documentation standard.

The style is lucid, the overall organization is well thought-out, and a sprinkling of humorous and pithy quotes help keep things in perspective. The suggested projects are appropriate, the exercises at the end of each chapter are worthwhile, and the references are excellent.

In addition to the standard topics of design, testing, maintenance, complexity metrics, etc., students will reap a harvest of wisdom and practical advice that only years of teaching combined with industry experience can produce. There are sections on group dynamics, scheduling problems, how to deal with customers who don't know what they want, human factors, sections on legal aspects of programming, software security, error trapping, and even a section, long overdue, on ethics.

The intended audience for this book is an undergraduate computer science student with a structured programming background that includes data structures and machine architecture. The authors indicate that the book could serve as a reference text for students after they graduate, and that it might be used for either a one-semester or two-semester course. There is plenty of material for two semesters, and there is room for a wide variety of teaching styles, but it might have been more helpful if a suggested breakdown of topics were offered for a two-semester course.

Professors who are teaching a software engineering course for the first time, and those with limited industry experience, should find the book particularly invaluable. (An instructor's guide is available, but not reviewed.)

Before choosing a software engineering text, it will be worthwhile to read the preface to this book, scan the table of contents, and read a dozen pages or so of the first chapter. The authors are interested in teaching software engineering rather than just writing a book, and it shows.

David J. Smith and Kenneth B. Wood. *Engineering Quality Software*

London: Elsevier Applied Science, 1989. ISBN 1-85166-358-4.

Table of Contents

Comment by Gary Ford, SEI

This book includes an extensive glossary and bibliographies of British and American standards for software products and processes.

Ian Sommerville. *Software Engineering, 3rd Ed.*

Reading, Mass.: Addison-Wesley Publishing Company, 1989. ISBN 0-201-17568-1. 653 pages. $36.75. Includes exercises.

Second edition reviewed in *Computing Reviews*, February 1989; *IEEE Software*, January 1987; *Software Engineering Journal*, July 1989.

Table of Contents

25. Project Planning and Scheduling
26. Software Cost Estimation
27. Software Maintenance
28. Configuration Management
29. Documentation
30. Software Quality Assurance

Review by Professor Frank Friedman, Temple University

This book is partitioned into an introductory section and five major parts, one each on Software Specification; Software Design; Programming Practice, Techniques and Environments; Software Validation; and Software Management. Ada is used as the example language throughout, but all material is understandable by any reader having a knowledge of Pascal, C, or Modula-2. Familiarity with the basics of programming techniques and data structures is required and the material on formal specification assumes a knowledge of elementary set theory. Aside from this, no mathematical background is required and few other assumptions about student background are made.

The author has a targeted audience of undergraduate and graduate students as well as practitioners; but despite its length, the book does not seem well suited to even most senior undergraduates. Most of the material is written at a very high, superficial level with insufficient detail and not enough substantial examples.

Those examples that are provided are rather small in scale. They are not effective in providing any sort of indication of larger scale system implications of the techniques and issues discussed in the text. While this is a problem with a number of introductory, survey kinds of texts, Sommerville's text suffers from it more so than most others.

For more sophisticated students, however—those with experience in working with large scale systems or those more skilled than the typical undergraduate at using supplementary references—this text may be more than adequate for an initial survey of the software engineering field. It covers a wide spectrum of material, including functional specification, the use of schemas, software design and implementation, programming environments (and CASE tools), verification, validation, testing and debugging, and a variety of management issues including costing, planning, configuration management, quality assurance and documentation. Obviously, none of these topics are covered in much depth.

Chapter summaries, extensive references and numerous small exercises are also provided. There are no case studies in the text and hence no large scale problems or projects are suggested. If appropriately supplemented, the book may be suitable for an introductory graduate-level survey course in software engineering, or as a good overview text for practitioners who might be new to the field.

Review by Professor Rodney L. Bown, University of Houston-Clear Lake

Sommerville’s third edition of *Software Engineering* is very readable and will support an introductory/survey course for third and/or fourth year students who have completed a course in the Ada language. This reviewer supports the choice of Ada as the language of choice.

In the preface and appendix, the author provides guidance on how to use the book as a course text to support several different courses. The size of the book is misleading (653 pages) in that the book has excessive white space. The problem with the book is that there are too many separate topics for one semester and not enough for two semesters. The treatment of each topic is shallow and requires that the instructor provide considerable support material. Chapters 7, 8, and 9 present topics which are related to formal methods. The coverage is too thin to support a complete teaching unit. Other examples would include the limited coverage in Chapter 11 on object-oriented design. This instructor uses supplementary material to support all teaching units.

The book does not have a consistent case study to demonstrate each of the software engineering phases. The book should not be a survey of different applications. Chapter 10 uses as examples: a control system, compiler, radar system, and spellcheck. The weather station in Chapter 11 is an excellent example but it is very incomplete. Chapter 11 also uses the heating system, address list, and counter as examples. The author should recognize that a third or fourth year university student has limited to no real-world application knowledge to appreciate multiple examples. My first recommendation for a fourth edition is to examine the positive results from using a consistent case study to support the ideas and concepts presented in the book.

The instructor's manual promised camera-ready figures to be used for production of visual aids by the instructor. This was a great disappointment in that the figures were not enlarged from those contained in the textbook. Many pages in the manual were used for programming lists and were of no value. The instructor's manual has provided limited support to this instructor.

The list of references is excellent.

Bottom Line: This instructor will continue to use the book with local supplementary material that supports a case study. This material includes but is not limited to topics such as IRDS, CAIS, DOD STD 2167, IEEE tutorial on Process Models, and brochures on commercial CASE products.

Donald W. Steward. *Software Engineering with Systems Analysis and Design*

Monterey, Calif.: Brooks/Cole Publishing Co., 1987. ISBN 0-534-07506-1.
414 pages. $44.75. Includes exercises.

Reviewed in *Computer*, April 1988.

Table of Contents

Ray Turner. *Software Engineering Methodology*

Reston, Va.: Reston Publishing Company, 1984. ISBN 0-8359-7022-1. 236 pages. $34.00.

Table of Contents

Anneliese von Mayrhauser. *Software Engineering: Methods and Management*

San Diego, Calif.: Academic Press, 1990. ISBN 0-12-727320-4. 864 pages. $49.95. Includes exercises.

Table of Contents

Richard Wiener and Richard Sincovec. *Software Engineering with Modula-2 and Ada*

New York: John Wiley & Sons, 1984. ISBN 0-471-89014-6. 451 pages. Includes exercises.

Reviewed in *Computing Reviews*, October 1985.

Table of Contents

6. Survey of Software Engineering Research Journals

Although the SEI curriculum recommendations are aimed at stimulating the growth of software engineering education, we hope that software engineering research will also become more widespread in universities. Faculty members often find that doing research and teaching in the same area improves their ability to do both.

A first step in initiating software engineering research is an awareness of the current research literature. Software engineering has matured to the point that there are now many research journals that report advances in the discipline. Many of these journals are relatively new and many professors are new to the discipline, so these journals may not be widely known. The two sections below identify archival journals in software engineering and journals in related areas that frequently include papers on software engineering. We recommend that schools developing software engineering degree programs ask their libraries to consider subscribing to these journals.

The information describing each journal has been extracted from the journal's information for authors and in most cases is a direct quotation.

6.1. Archival Journals

ACM Transactions on Software Engineering and Methodology

Publisher: Association for Computing Machinery.

This new journal (first issue in January 1991) will publish significant papers on all aspects of research related to complex software systems, generally characterized by a scale requiring development by teams, not individuals. It will include research issues such as specific tools, methods, technologies, environments, and platforms; languages, algorithms data structures, theory as appropriate to the scale and complexity of the application domains and within context of large-scale systems; results which are extensible, scalable with practical import; work demonstrating clear application of scientific method: thesis, development, experiment, evolution and iteration; empirical studies; application of innovative technologies, such as: data and object management; intelligent systems and artificial intelligence; new programming styles; new hardware, systems, and graphics; methods for real-time and other time-constrained systems; techniques for prototyping and reuse; and risk evaluations and containment.

Formal Aspects of Computing

Publisher: British Computer Society.

Computing science is developing and providing a basis on which complex systems can be designed and analysed. Theories are evolving in terms of which a true understanding of difficult computing concepts can be gained. To employ such theories in discussions requires the use of formal notation. Although notation can present an initial barrier, practitioners are now finding that the investment of effort is worthwhile.

The principal aims of this journal are to promote the growth of computing science, to show its relationship to practice, and to help in the application of formalisms. In particular, contributions to the formal aspects of computing are to be published. The following fall within the scope of formal aspects: Well founded notations for system description/specification, verifiable designs, proof methods, theories of objects used in specifications and implementations, transformational design, formal approaches to requirements analysis, results on algorithm and problem complexity, fault-tolerant design, descriptions of relevant “project support environments,” methods of approaching development.

Applications of known formal methods as well as new results would be suitable subjects for papers. Comprehensive surveys will also be published and there is hope that some systematic coverage of major topics can be achieved over a period of years. Contributions to the teaching of formal aspects would also be welcome.

IEEE Transactions on Software Engineering

Publisher: Institute for Electrical and Electronic Engineers.

The *IEEE Transactions on Software Engineering* is an archival journal published monthly. We are interested in well-defined theoretical results and empirical studies that have potential impact on the construction, analysis, or management of software. The scope of the *Transactions* ranges from the theoretical to the practical. We welcome treatments ranging from formal models to empirical studies, from software constructive paradigms to assessment mechanisms, from the development of principles to the application of those principles to specific environments. Since the journal is archival, it is assumed that the ideas presented are important, have been well analyzed and/or empirically validated and are of value to the software engineering research or practitioner community.

Specific topic areas include: a) development and maintenance methods and models, e.g., techniques and principles for the specification, design, implementation of software systems including notations and process models; b) assessment methods, e.g., software tests and validation, reliability models, test and diagnosis procedures, software redundancy and design for error control, and the measurements and evaluation of various aspects of the process and product; c) software project management, e.g., productivity factors, cost models, schedule and organizational issues, standards;

d) tools and environments, e.g., specific tools, integrated tool environments including the associated architectures, databases, and parallel and distributed processing issues; e) system issues, e.g., hardware-software tradeoffs; and f) state-of-the-art surveys that provide a synthesis and comprehensive review of the historical development of one particular area of interest.

Information and Software Technology

Publisher: Butterworth Heinemann, 80 Montvale Avenue, Stoneham, MA 02180.

Information and Software Technology focuses on international research on software development and its application in industry. The journal brings the results of government projects and academic and commercial research to its readers. It discusses techniques for tailoring software and information systems for institutional, industrial and commercial use, and it explains in depth the technical needs of its users for more efficient and reliable systems. Emphasis is placed on the use of proper methodologies and formal practices to improve software productivity.

Among the topics covered are software engineering development standards, tools and methodologies, environments, metrics, automatic program development, quality assurance and testing, formal methods, project management, and reliability and maintenance.

International Journal of Computer and Software Engineering

Publisher: Ablex Publishing Corporation, 355 Chestnut Street, Norwood, NJ 07648. (The journal is scheduled to begin publication in 1991.)

The *International Journal of Computer and Software Engineering* will present state-of-the-art research and development in the area of computer engineering and software engineering. The journal will publish refereed papers relating to a wide variety of topics, theory, and practical applications in these areas.

The journal will be of interest to all those concerned with research and development, especially engineers, computer scientists, and those in allied disciplines. It will publish original research, development, and state-of-the-art tutorials and related topics. All papers submitted to the journal will be refereed by an international review board.

Journal of Software Maintenance: Research and Practice

Publisher: John Wiley & Sons Ltd., Baffins Lane, Chichester, Sussex PO19 1UD, England.

The *Journal of Software Maintenance* publishes refereed papers in all aspects of software maintenance. We seek to include articles from practitioners working in the field (including the user community) as well as research papers. It is not the intention to publish papers on software development except where topics directly of relevance to maintenance are addressed. These could include (for example) the production of more maintainable software, or metrics produced during development to predict the maintainability, quality or reliability of software. Papers on the impact for maintenance of new software practices will be welcomed.

Other journals do not cater well for "post-delivery" and operational support issues, and the *Journal of Software Maintenance* fills this need. The editors' aim is to convey the results of academic research and practical experience into the computing community.

Topics on which papers will be published include: software evolution lifecycles, software maintenance management, tools, environments, metrics and productivity methods, quality assurance, theory of software maintenance, maintainability of new software, methods for software maintenance, impact for maintenance of new software practices.

Journal of Systems and Software

Publisher: Elsevier Science Publishing Co., Inc., 655 Avenue of the Americas, New York, NY 10010.

The *Journal of Systems and Software* publishes papers covering all aspects of programming methodology, software engineering, and related hardware-software-systems issues. Topics of interest include, but are not limited to, software tools, programming environments, techniques for developing, validating, and maintaining software systems, prototyping issues, high level specification techniques, procedural and functional programming techniques, data flow concepts, multiprocessing, real-time, distributed concurrent, and telecommunications systems, software metrics, reliability models for software, performance issues, and management concerns. The journal publishes research papers, state-of-the-art surveys, and reports of practical experience.

Journal of Systems Integration

Publisher: Kluwer Academic Publishers, 101 Philip Drive, Norwell, MA 02061-1677.

The *Journal of Systems Integration* is a quarterly peer-reviewed technical publication containing original, survey application, and research papers on all topics related to systems integration. The intent is to encompass a collection of papers that heretofore have been dispersed throughout a wide body of literature involving the interaction of disciplines, technologies, methods and machines necessary to integrate various constituent systems.

The scope of this journal generally parallels the definition of the integration of computer systems. However, it also deals with the general integration of processes and systems, and the development of mechanisms and tools enabling solutions to multi-disciplinary problems found in the computer services and manufacturing industries. This journal focuses on the following critical steps found in effective systems integrations: process characterization, re-engineering and simplification of processes, convergence on a common system architecture, and automation of the processes and systems.

Since the successful implementation of these steps for systems integration requires diverse knowledge bases and expertise in various areas, the journal also emphasizes additional topics such as managing knowledge and information that are physically distributed in various databases; computer communications impact on the system process; dealing with heterogeneous computers and environments, and coordinating diverse computer communications networks with information networks.

Real-Time Systems

Publisher: Kluwer Academic Publishers, 101 Philip Drive, Norwell, MA 02061-1677.

Real-time systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced. *Real-Time Systems* is an archival, peer-reviewed, technical journal publishing the following types of papers, which concentrate on real-time computing principles and applications: research papers, invited papers, reports on projects and case studies, standards and corresponding proposals for general discussion, and a partitioned tutorial on real-time systems as a continuing series.

Software Engineering Journal

Publisher: The Institution of Electrical Engineers (IEE), Savoy Place, London WC2R 0BL, United Kingdom.

Software Engineering Journal welcomes original contributions of interest to practitioners, researchers and managers engaged in software engineering. It is intended that the journal should cover the spectrum from reports on practical experience using software engineering methods and tools through to papers on longterm research activities. The Editors particularly welcome material which is of practical benefit today, or which could be brought into practical application in the foreseeable future.

The scope of the journal is the whole of software engineering, including, but not limited to: management of software development; the development process; standards; support environments (IPSEs); development methods and tools; formal methods; software engineering training; software metrics and estimation methods; high-reliability systems.

Software: Practice and Experience

Publisher: John Wiley & Sons Ltd., Baffins Lane, Chichester, Sussex PO19 1UD, England.

To write successful software requires a great deal of know-how, and far too little of this is at present available in written form, which results in a duplication of inventive effort. Other journals do not cater well for the software writer and *Software* fills this need; the emphasis is on conveying the results of practical experience for the benefit of the computing community. Both “systems” software and “applications” software, for use in batch, multi-access, interactive and real-time environments are included. Articles cover software design and implementation, case studies which describe the evolution of systems and the thinking behind them, and critical appraisals of software systems. Well-tried techniques that are not documented are included in articles of a tutorial nature. Although the emphasis is on practical experience, articles of a theoretical or mathematical nature will be included when it is felt that an understanding of theory will lead to better practical systems.

6.2. Other Journals

In addition to the journals that focus primarily on software engineering, there are several others that frequently include software engineering papers. These include:

ACM Transactions on Programming Languages and Systems

Publisher: Association for Computing Machinery.

The purpose of the *ACM Transactions on Programming Languages and Systems* (TOPLAS) is to provide a unified forum for the presentation of research and development results on all aspects of the design, definition, realization, and use of programming languages and systems. In addition to the traditional research contributions, the journal includes reports of the insights and experience gained by the practitioner in applying the fruits of that research.

Among the topics within the scope of TOPLAS are design or comparisons of language features, analysis of the design or implementation of particular languages, description of novel program development tools and techniques, testing and verification methodology, user experience with languages or methodologies, program specification languages and methods, storage allocation and garbage collection algorithms, methods for specifying programming language semantics, language constructs for and programming of asynchronous and distributed processes, and applications of programming languages or methods to other areas such as databases and office automation.

Communications of the ACM

Publisher: Association for Computing Machinery.

Communications of the ACM is a monthly magazine that publishes articles of general interest to the computing community. Papers of direct and immediate interest to practitioners are published in the Computing Practices section. Submitted manuscripts should emphasize the skills and techniques used daily, including the design and construction of applications systems; discussion of computer systems and tools; methodologies for management; computer philosophy; and the implications of theoretical contributions to application areas. Papers submitted to Computing Practices are reviewed by the Computing Practices Panel for content, definitiveness, interest, and importance to the practice of computing.

Case Studies are articles that report on experiences gained and lessons learned constructing and using major computer systems. They take a comprehensive view of selected systems, covering them from requirements through design, implementation, and use. Case Studies should take a rigorously objective perspective on the systems they describe, and should be both evaluative and descriptive.

Computer

Publisher: Institute for Electrical and Electronic Engineers.

Computer covers all aspects of computer science, engineering, technology, and applications. It is aimed at a broad audience. Articles in *Computer* are usually survey or tutorial in nature and cover the state of the art or important emerging developments. *Computer* publishes technically substantive articles that are referenced extensively in the literature. One of the important purposes of *Computer* is to act as a technology transfer conduit to bring results and formalisms from university, industry and government research and development centers to the general practitioners in the field.

Computer Journal

Publisher: British Computer Society.

In order to maintain the *Journal* as a leading medium in all aspects of computer science and compute applications, it has been agreed that future issues will identify state-of-the-art themes and will publish new papers related to those themes. By "new" it is intended that papers would have been written no longer than six months before the press date. The list of themes to be chosen will be published approximately 18 months in advance and submissions will be invited against the themes.

Future special issues include Concurrent Programming for the February 1991 issue and Methodologies (Software and Systems) for the April 1991 issue.

IEEE Software

Publisher: Institute for Electrical and Electronic Engineers.

IEEE Software emphasizes current practice and experience together with promising new ideas that are likely to be used in the near term. It is directed to the practice of the software profession. We welcome papers on topics across the software spectrum. Sample topics are data engineering and database software; programming environments; languages and language-related issues; knowledge-based and decision support systems; program and system debugging and testing; distributed, network, and parallel systems; education of software professionals; design, development, and programming methodologies; algorithms, their analysis and pragmatics;

performance measurement and evaluation; program and system reliability, security, and verification; software-related social and legal issues; and human factors and software metrics.

Articles describing how software is developed in specific companies, laboratories, and university environments, as well as articles describing new tools to aid in the software and system development process, are welcome. The intent is to provide the reader with information on advances in software technology, specifics on novel features and applications, contrasts in designing and programming in the large versus designing and programming in the small, discussions of limitations and failures, and an awareness of the trends in this rapidly evolving area. Tutorials, survey articles, standards, guided tours through descriptions of projects, designs, or algorithms, and case studies are also encouraged.

IEEE Transactions on Engineering Management

Publisher: Institute for Electrical and Electronic Engineers.

IEEE Transactions on Engineering Management is a research-based, refereed journal in engineering management. The purpose of this *Transactions* is multifold: to assist in the establishment and recognition of the engineering management discipline; to provide the publications medium for authors at the leading edge of engineering management in academic institutions, industrial organizations, government agencies, or other settings; to establish the guidelines and identify the future directions of critical issues in engineering management; and to become a forum for the researchers and practitioners of engineering management, regardless of their technical specialties.

IEEE Transactions on Parallel and Distributed Systems

Publisher: Institute for Electrical and Electronic Engineers.

The goal of the *Transactions* is to publish a range of papers, short notes, and survey articles that deal with the research areas of current importance to our readers. Current areas of particular interest include but are not limited to: a) architectures—design, analysis, and implementation of multiple-processor systems (including multiprocessors, multicomputers, and networks); impact of VLSI on system design; interprocessor communications; b) software—parallel languages and compilers; scheduling and task partitioning, databases, operating systems, and programming environments for multiple-processor systems; c) algorithms and applications—models of computation; analysis and design of parallel/distributed algorithms; application studies resulting in better multiple-processor systems; d) other issues—performance measurements, evaluation, modeling and simulation of multiple-processor systems; real-time, reliability and fault-tolerance issues; conversion of software from sequential-to-parallel forms.

Interacting with Computers

Publisher: Butterworth Heinemann Publishers.

The objective of the study of human-computer interaction (HCI) is to understand how people communicate with computers so that the design and use of computer systems can be improved. HCI is an increasingly important, but diverse and fragmented field. A new medium is needed to allow the communication and coordination of expertise, findings and activities. *Interacting with Computers* is the international forum for just such communication. It makes information accessible not only to those engaged in research but to all HCI practitioners.

Coverage includes: systems and dialogue design; evaluation techniques; user interface design, tools and methods; empirical evaluations; user features and user modelling; new research paradigms; design theory, process and methodology; organizational and societal issues; intelligent systems; training and education applications; and emerging technologies.

Journal of Parallel and Distributed Computing

Publisher: Academic Press.

Among the areas in which papers are solicited are software tools and environments, real-time systems, performance analysis, and fault-tolerant computing.

Structured Programming

Publisher: Springer International.

The international journal *Structured Programming* will serve the professional computing and engineering community. Its scope will include technical contributions and short communications in the areas of programming, programming methodology and style, programming languages, programming environments, compilers, interpreters, and applications. It will report on technical advances in the field, announce and review systems, implementations, and relevant publications. *Structured Programming* will emphasize innovative concepts in programming (such as literate programming) and practical solutions to real problems. *Structured Programming* is not intended as an archival journal, but rather an informal forum for the timely exchange of ideas and information.

Appendix 1. An Organizational Structure for Curriculum Content

The body of knowledge called *software engineering* consists of a large number of interrelated topics. We thought it impractical to attempt to capture this knowledge as an undifferentiated mass, so an organizational structure was needed. The structure described below *is not intended to be a taxonomy of software engineering.* Rather, it is a guide that helps the SEI to collect and document software engineering knowledge and practice, and to describe the content of some recommended courses for a graduate curriculum.

Discussions of software engineering frequently describe the discipline in terms of a software life cycle: requirements analysis, specification, design, implementation, testing, and maintenance. Although these life cycle phases are worthy of presentation in a curriculum, we found this one-dimensional structure inadequate for organizing all the topics in software engineering and for describing the curriculum.

A good course, whether a semester course in a university or a one-day training course in industry, must have a central thread or idea around which the presentation is focused. Not every course can or should focus on one life cycle phase. In an engineering course (including software engineering), we can look at either the engineering *process* or the *product* that is the result of the process. Therefore, we have chosen these two views as the highest level partition of the curriculum content. Each is elaborated below.

The Process View

The process of software engineering includes several *activities* that are performed by software engineers. The range of activities is broad, but there are many *aspects* of each activity that are similar across that range. Thus, we organize those topics whose central thread is the process in two dimensions: activity and aspect (see Figure A1.1).

The Activity Dimension

Activities are divided into four groups: *development, control, management, and operations.* Each is defined and discussed below.

Development activities are those that create or produce the artifacts of a software system. These include requirements analysis, specification, design, implementation, and testing. Because a software system is usually part of a larger system, we sometimes distinguish system activities from software activities; for example, system design from software design. We expect that many large projects will include both systems engineers and software engineers, but it is important for software engineers to appreciate the systems aspects of the project, and system activities should be included in a curriculum.

Control activities are those that exercise restraining, constraining, or directing influence over software development. These activities are more concerned with controlling the way in which the development activities are performed than with producing artifacts. Two major kinds of control activities are those related to software evolution and those related to software quality.

A software product evolves in the sense that it exists in many different forms as it moves through its life cycle, from initial concept, through development and use, to eventual retirement. Change control and configuration management are activities related to evolution. We also consider software maintenance to be in this category, rather than as a separate development activity, because the difference between development and maintenance is not in the activities performed (both involve requirements analysis, specification, design, implementation, and testing), but in the way those activities are constrained and controlled. For example, the fundamental constraint in software maintenance is the pre-existence of a software system coupled with the belief that it is more cost-effective to modify that system than to build an entirely new one.

Software quality activities include quality assurance, test and evaluation, and independent verification and validation. These activities, in turn, incorporate such tasks as software technical reviews and performance evaluation.

Management activities are those involving executive, administrative, and supervisory direction of a software project, including technical activities that support the executive decision process. Typical management activities are project planning (schedules, establishment of milestones), resource allocation (staffing, budget), development team organization, cost estimation, and handling legal concerns (contracting, licensing). This is an appropriate part of a software engineering curriculum for several reasons: there is a body of knowledge about managing software projects that is different from that about managing other kinds of projects; many software engineers are likely to assume software management positions at some point in their careers; and knowledge of this material by all software engineers improves their ability to work together as a team on large projects.

Operations activities are those related to the use of a software system by an organization. These include training personnel to use the system, planning for the delivery and installation of the system, transition from the old (manual or automated) system to the new, operation of the software, and retirement of the system. Although software engineers may not have primary responsibility for any of these activities, they are often participants on teams that perform them. Moreover, an awareness of these activities will often affect the choices software engineers make during the development of a system.

The operation of software engineering support tools provides a case of special interest. These tools are software systems, and the users are the software engineers themselves. Operations activities for these systems can be observed and experienced directly. An awareness of the issues related to the use of software tools can help software engineers not only develop systems for others but also adopt and use new tools for their own activities.

The Aspect Dimension

Engineering activities traditionally have been partitioned into two categories: *analytic* and *synthetic*. We have chosen instead to consider an axis orthogonal to activities that captures some of this kind of distinction, but that recognizes six *aspects* of these activities: *abstractions, representations, methods, tools, assessment,* and *communication*.

Abstractions include fundamental principles and formal models. For example, software development process models (waterfall, iterative enhancement, etc.) are models of software evolution. Finite state machines and Petri nets are models of sequential and concurrent computation, respectively. COCOMO is a software cost estimation model. Modularity and information hiding are principles of software design.

Representations include notations and languages. The Ada programming language thus fits into the organization as an implementation language, while decision tables and data flow diagrams are design notations. PERT charts are a notation useful for planning projects.

Methods include formal methods, current practices, and methodologies. Proofs of correctness are examples of formal methods for verification. Object-oriented design is a design method, and structured programming can be considered a current practice of implementation.

Tools include individual software tools as well as integrated tool sets (and, implicitly, the hardware systems on which they run). Examples are general-purpose tools (such as electronic mail and word processing), tools related to design and implementation (such as compilers and syntax-directed editors), and project management tools. Other types of software support for process activities are also included; these are sometimes described by such terms as *infrastructure, scaffolding,* or *harnesses*. Sometimes the term *environment* is used to describe a set of tools, but we prefer to reserve this term to mean a collection of related representations, tools, methods, and objects. Software objects are abstract, so we can only manipulate representations of them. Tools to perform manipulations are usually designed to help automate a particular method or way of accomplishing a task. Typical tasks involve many objects (code modules, requirements specification, test data sets, etc.), so those objects must be available to the tools. Thus, we believe all four—representations, tools, methods, and objects—are necessary for an environment.

Assessment aspects include measurement, analysis, and evaluation of both software products and software processes, and of the impact of software on organizations. Metrics and standards are also placed in this category. This is an area we believe should be emphasized in the curriculum. Software engineers, like engineers in more traditional fields, need to know what to measure, how to measure it, and how to use the results to analyze, evaluate, and ultimately improve processes and products.

Communication is the final aspect. All software engineering activities include written and oral communication. Most produce documentation. A software engineer must have good general technical communication skills, as well as an understanding of forms of documentation appropriate for each activity.

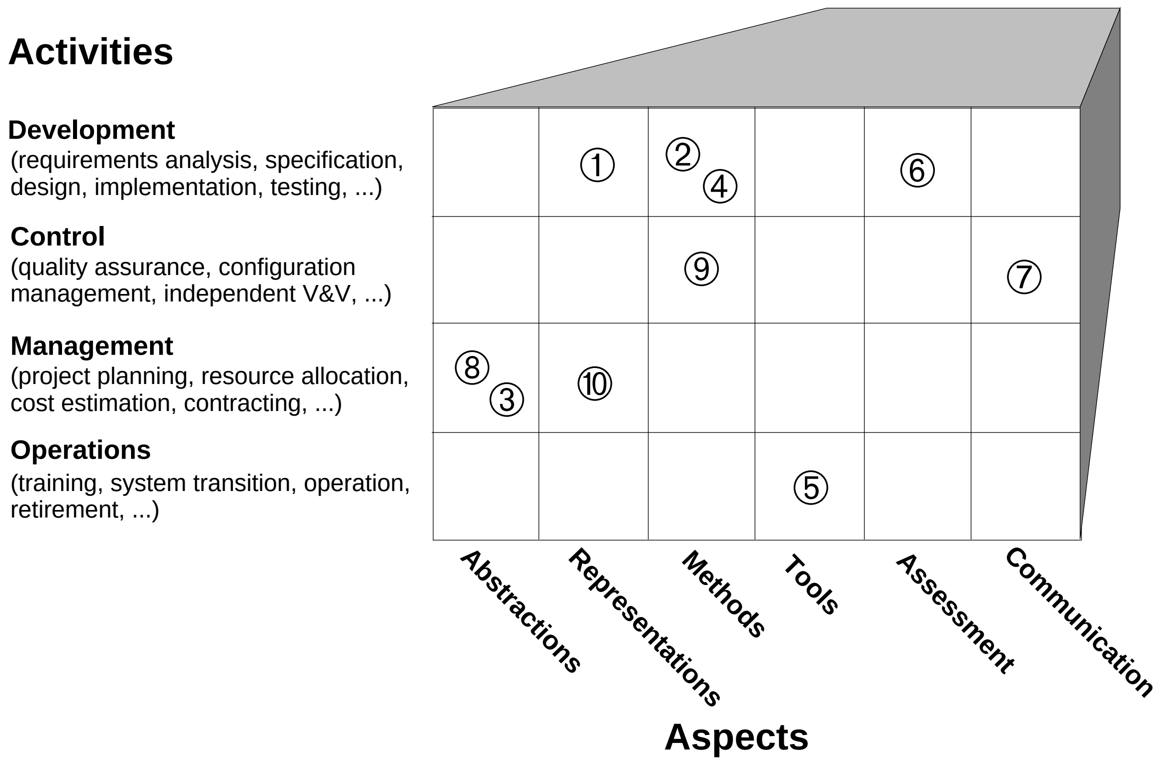
By considering the activity dimension and the aspect dimension as orthogonal, we have a matrix of ideas that might serve as the central thread in a course (Figure A1.1). It is likely that individual cells in the matrix represent too specialized a topic for a full semester course. Therefore, we recommend that courses be designed around part or all of a horizontal or vertical slice through that matrix.

The Product View

Often it is appropriate to discuss many activities and aspects in the context of a particular kind of software system. For example, concurrent programming has a variety of notations for specification, design, and implementation that are not needed in sequential programming. Instead of inserting one segment or lecture on concurrent programming in each of several courses, it is probably better to gather all the appropriate information on concurrent programming into one course. A similar argument can be made for information related to various system requirements; for example, achieving system robustness involves aspects of requirements definition, specification, design, and testing.

Therefore we have added two additional categories to the organizational structure for curriculum content: *software system classes* and *pervasive system requirements*. Although these may be viewed as being dimensions orthogonal to the activity and aspect dimensions, it is not necessarily the case that every point in the resulting four-dimensional space represents a topic for which there exists a body of knowledge, or for which a course should be taught. Figure A1.2 shows an example of a point for which there is probably a very small but nonempty body of knowledge.

Any of the various system classes or pervasive requirements described below might be the central thread in a course in a software engineering curriculum. We emphasize that the same material might be taught in courses whose central thread is one of the activities mentioned earlier. For example, techniques for designing real-time systems could be taught in a design course or in a real-time systems course. Testing methods to achieve system robustness could be taught in a testing course or in a robustness course. The purpose of adding these two new dimensions to the structure is to allow better descriptions of possible courses.


Activities
Development
(requirements analysis, specification, design, implementation, testing, ...)
Control
(quality assurance, configuration management, independent V&V, ...)
Management
(project planning, resource allocation, cost estimation, contracting, ...)
Operations
(training, system transition, operation, retirement, ...)
①
②
④
⑥
⑨
⑦
⑧
③
⑩
⑤
Abstractions
Representations
Methods
Tools
Assessment
Communication
Aspects

Examples

1.	Ada	6.	Performance Evaluation
2.	Object-Oriented Design	7.	Configuration Management Plan
3.	COCOMO Model	8.	Waterfall Model
4.	Path Coverage Testing	9.	Code Inspection
5.	Interactive Video	10.	PERT Chart

Figure A1.1. The process view: examples of activities and aspects

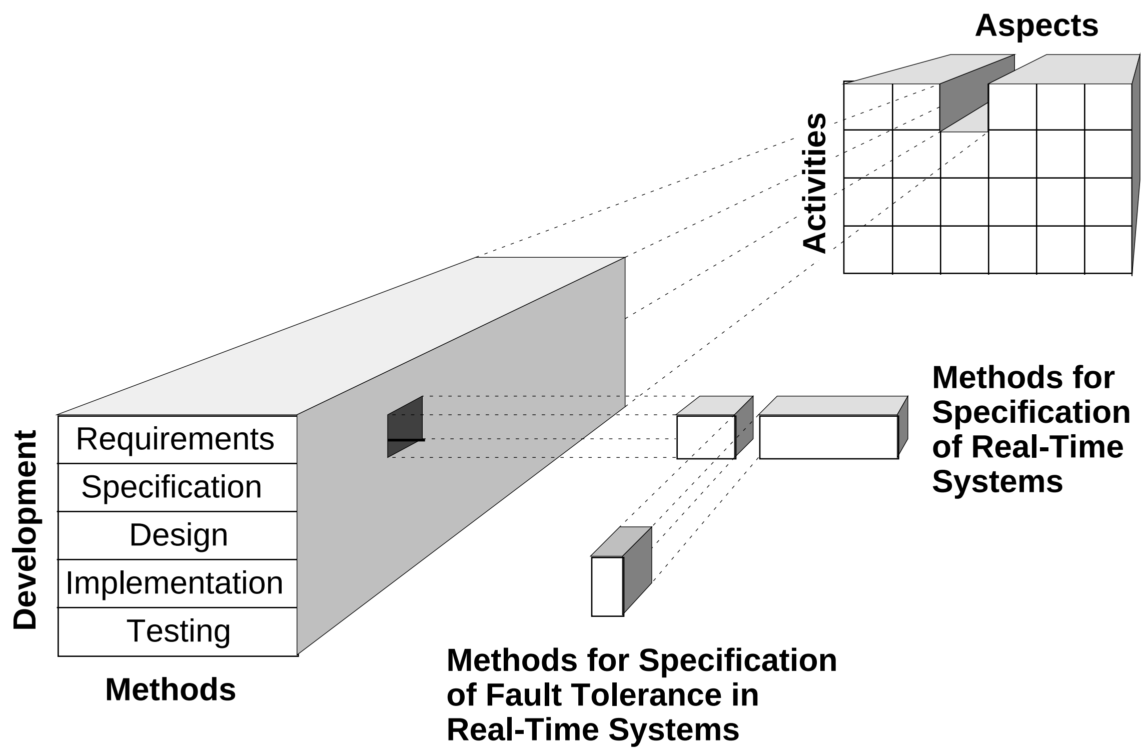

Aspects
Activities
Methods for Specification of Real-Time Systems
Development
Requirements
Specification
Design
Implementation
Testing
Methods
Methods for Specification of Fault Tolerance in Real-Time Systems

Figure A1.2. Organizational structure for curriculum content

Software System Classes

Several different classes can be considered. One group of classes is defined in terms of a system's relationship to its environment; its members are described by terms such as *batch*, *interactive*, *reactive*, *real-time*, and *embedded*. Another group has members described by terms such as *distributed*, *concurrent*, or *network*. Another has members defined in terms of internal characteristics, such as *table-driven*, *process-driven*, or *knowledge-based*. We also include generic or specific applications areas, such as *avionics systems*, *communications systems*, *operating systems*, or *database systems*.

Clearly, these classes are not disjoint. Each class is composed of members that have certain common characteristics, and there is or may be a body of knowledge that directly addresses the development of systems with those characteristics. Thus each class may be the central theme in a software engineering course.

Pervasive System Requirements

Discussions of system requirements generally focus on functional requirements. There are many other categories of requirements that also deserve attention. Identifying and then meeting those requirements is the result of many activities performed throughout the software engineering process. As with system classes, it may be appropriate to choose one of these requirement categories as the central thread for a course, and then to examine those activities and aspects that affect it.

Examples of pervasive system requirements are *accessibility, adaptability, availability, compatibility, correctness, efficiency, fault tolerance, integrity, interoperability, maintainability, performance, portability, protection, reliability, reusability, robustness, safety, security, testability,* and *usability*. Definitions of these terms may be found in the ANSI/IEEE Glossary of Software Engineering Terminology [IEEE83].

Appendix 2. Bloom's Taxonomy of Educational Objectives

Bloom [Bloom56] has defined a *taxonomy of educational objectives* that describes several levels of knowledge, intellectual abilities, and skills that a student might derive from education. An adaptation of this taxonomy for software engineering is shown below. This taxonomy can be used to help describe the objectives—and thus the style and depth of presentation—of a software engineering curriculum.

Evaluation: The student is able to make qualitative and quantitative judgments about the value of methods, processes, or artifacts. This includes the ability to evaluate conformance to a standard, and the ability to develop evaluation criteria as well as apply given criteria. The student can also recognize improvements that might be made to a method or process, and to suggest new tools or methods.

Synthesis: The student is able to combine elements or parts in such a way as to produce a pattern or structure that was not clearly there before. This includes the ability to produce a plan to accomplish a task such that the plan satisfies the requirements of the task, as well as the ability to construct an artifact. It also includes the ability to develop a set of abstract relations either to classify or to explain particular phenomena, and to deduce new propositions from a set of basic propositions or symbolic representations.

Analysis: The student can identify the constituent elements of a communication, artifact, or process, and can identify the hierarchies or other relationships among those elements. General organizational structures can be identified. Unstated assumptions can be recognized.

Application: The student is able to apply abstractions in particular and concrete situations. Technical principles, techniques, and methods can be remembered and applied. The mechanics of the use of appropriate tools have been mastered.

Comprehension: This is the lowest level of understanding. The student can make use of material or ideas without necessarily relating them to others or seeing the fullest implications. Comprehension can be demonstrated by rephrasing or translating information from one form of communication to another, by explaining or summarizing information, or by being able to extrapolate beyond the given situation.

Knowledge: The student learns terminology and facts. This can include knowledge of the existence and names of methods, classifications, abstractions, generalizations, and theories, but does not include any deep understanding of them. The student demonstrates this knowledge only by recalling information.

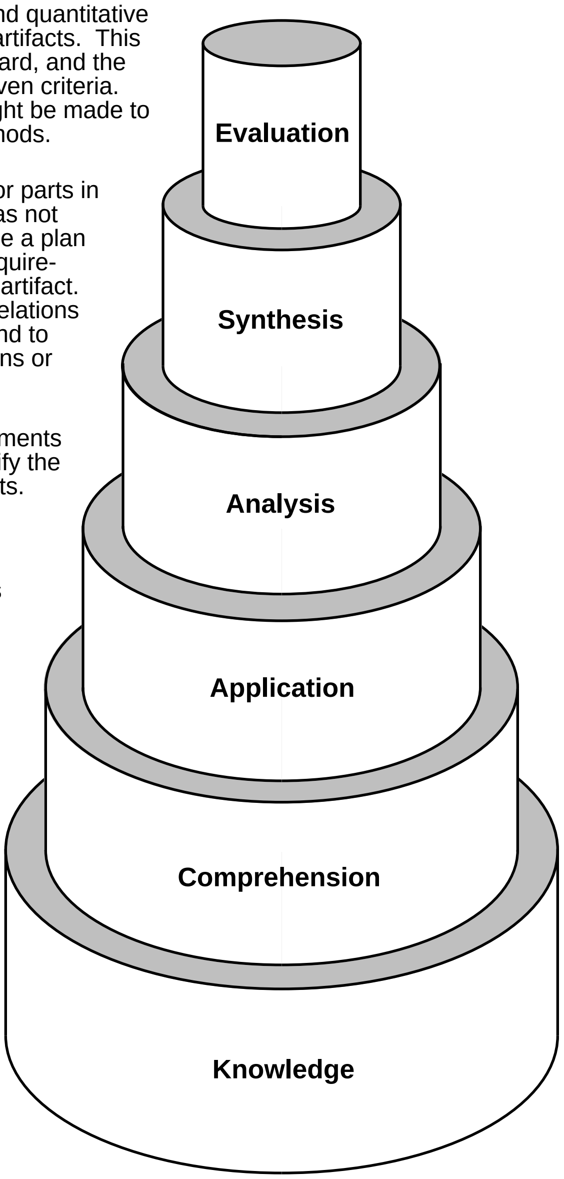


Figure A2.1. Bloom's taxonomy of educational objectives

Appendix 3. SEI Curriculum Modules and Other Publications

The SEI Education Program has produced a variety of educational materials to support software engineering education. Those documents in the list below that have DTIC numbers (of the form ADA000000) are available from the Defense Technical Information Center (DTIC) and the National Technical Information Service (NTIS). Other documents (excluding conference proceedings) are available from the SEI; please send *written* requests, accompanied by a mailing label, to the Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, Attn.: Publications Requests.

SEI Curriculum Modules and Support Materials

A *curriculum module* documents and explicates a body of knowledge within a relatively small and focused topic area of software engineering. Its major components are a detailed, annotated outline of the topic area, an annotated bibliography of the important literature in the area, and suggestions for teaching the material. A module is intended to be used primarily by an instructor in designing and teaching part or all of a course.

A *support materials* package includes a variety of materials helpful in teaching a course, such as examples, exercises, or project ideas. Contributions from software engineering educators are solicited.

The currently available modules and support materials packages are listed below.[†] For each module, a *capsule description*, which is similar to a college catalog description or the abstract of a technical paper, is included.

Introduction to Software Design

David Budgen,
University of Stirling

SEI-CM-2-2.1

This curriculum module provides an introduction to the principles and concepts relevant to the design of large programs and systems. It examines the role and context of the design activity as a form of problem-solving process, describes how this is supported by current design methods, and considers the strategies, strengths, limitations, and main domains of application of these methods.

[†]CM-1, CM-15, and CM-18 do not appear in this list. CM-1 has been superseded by CM-19, CM-15 is still under development, and CM-18 has been superseded by CM-23.

The Software Technical Review Process

James Collofello,
Arizona State University

SEI-CM-3-1.5

This curriculum module consists of a comprehensive examination of the technical review process in the software development and maintenance life cycle. Formal review methodologies are analyzed in detail from the perspective of the review participants, project management and software quality assurance. Sample review agendas are also presented for common types of reviews. The objective of the module is to provide the student with the information necessary to plan and execute highly efficient and cost effective technical reviews.

Support Materials for The Software Technical Review Process

Edited by John Cross,
Indiana University of Pennsylvania

SEI-SM-3-1.0

This support materials package includes materials helpful in teaching a course on the software technical review process.

Software Configuration Management

James E. Tomayko,
The Wichita State University

SEI-CM-4-1.3

Software configuration management encompasses the disciplines and techniques of initiating, evaluating, and controlling change to software products during and after the development process. It emphasizes the importance of configuration control in managing software production.

Support Materials for Software Configuration Management

Edited by James E. Tomayko, The Wichita State University

SEI-SM-4-1.0

This support materials package includes materials helpful in teaching a course on configuration management.

Information Protection

Fred Cohen,
University of Cincinnati

SEI-CM-5-1.2

This curriculum module is a broad based introduction to information protection techniques. Topics include the history and present state of cryptography, operating system protection, network protection, data base protection, physical security techniques, cost benefit tradeoffs, social issues, and current research trends. The successful student in this course will be prepared for an in-depth course in any of these topics.

Software Safety

Nancy Leveson,
University of California, Irvine

SEI-CM-6-1.1

Software safety involves ensuring that software will execute within a system context without resulting in unacceptable risk. Building safety-critical software requires special procedures to be used in all phases of the software development process. This module introduces the problems involved in building such software along with the procedures that can be used to enhance the safety of the resulting software product.

Assurance of Software Quality

Brad Brown,
Boeing Military Airplanes

SEI-CM-7-1.1

This module presents the underlying philosophy and associated principles and practices related to the assurance of software quality. It includes a description of the assurance activities associated with the phases of the software development life-cycle (*e.g.*, requirements, design, test, etc.).

Formal Specification of Software

Alfs Berztiss,
University of Pittsburgh

SEI-CM-8-1.0

This module introduces methods for the formal specification of programs and large software systems, and reviews the domains of application of these methods. Its emphasis is on the functional properties of software. It does not deal with the specification of programming languages, the specification of user-computer interfaces, or the verification of programs. Neither does it attempt to cover the specification of distributed systems.

Support Materials for Formal Specification of Software

Edited by Alfs Berztiss,
University of Pittsburgh

SEI-SM-8-1.0

This support materials package includes materials helpful in teaching a course on formal specification of software.

Unit Testing and Analysis

Larry Morell,
College of William and Mary

SEI-CM-9-1.2

This module examines the techniques, assessment, and management of unit testing and analysis. Testing and analysis strategies are categorized according to whether their coverage goal is functional, structural, error-oriented, or a combination of these. Mastery of the material in this module allows the software engineer to define, conduct, and evaluate unit tests and analyses and to assess new techniques proposed in the literature.

Models of Software Evolution: Life Cycle and Process

Walt Scacchi,
University of Southern California

SEI-CM-10-1.0

This module presents an introduction to models of software system evolution and their role in structuring software development. It includes a review of traditional software life-cycle models as well as software process models that have been recently proposed. It identifies three kinds of alternative models of software evolution that focus attention to either the products, production processes, or production settings as the major source of influence. It examines how different software engineering tools and techniques can support life-cycle or process approaches. It also identifies techniques for evaluating the practical utility of a given model of software evolution for development projects in different kinds of organizational settings.

Software Specification: A Framework

Dieter Rombach,
University of Maryland

SEI-CM-11-2.1

This module provides a framework for specifying software processes and products. The specification of a software product type describes how the corresponding products should look. The specification of a software process type describes how the corresponding processes should be performed.

Software Metrics

Everald Mills,
Seattle University

SEI-CM-12-1.1

Effective management of any process requires quantification, measurement, and modeling. Software metrics provide a quantitative basis for the development and validation of models of the software development process. Metrics can be used to improve software productivity and quality. This module introduces the most commonly used software metrics and reviews their use in constructing models of the software development process. Although current metrics and models are certainly inadequate, a number of organizations are achieving promising results through their use. Results should improve further as we gain additional experience with various metrics and models.

Introduction to Software Verification and Validation

James Collofello,
Arizona State University

SEI-CM-13-1.1

Software verification and validation techniques are introduced and their applicability discussed. Approaches to integrating these techniques into comprehensive verification and validation plans are also addressed. This curriculum module provides an overview needed to understand in-depth curriculum modules in the verification and validation area.

Intellectual Property Protection for Software

Pamela Samuelson and Kevin Deasy, University of Pittsburgh School of Law

SEI-CM-14-2.1

This module provides an overview of the U.S. intellectual property laws that form the framework within which legal rights in software are created, allocated, and enforced. The primary forms of intellectual property protection that are likely to apply to software are copyright, patent, and trade secret laws, which are discussed with particular emphasis on the controversial issues arising in their application to software. A brief introduction is also provided to government software acquisition regulations, trademark, trade dress, and related unfair competition issues that may affect software engineering decisions, and to the Semiconductor Chip Protection Act.

Software Development Using VDM

Jan Storbank Pedersen, Dansk Datamatik Center

SEI-CM-16-1.1

This module introduces the Vienna Development Method (VDM) approach to software development. The method is oriented toward a formal model view of the software to be developed. The emphasis of the module is on formal specification and systematic development of programs using VDM. A major part of the module deals with the particular specification language (and abstraction mechanisms) used in VDM.

User Interface Development

Gary Perlman, Ohio State University

SEI-CM-17-1.1

This module covers the issues, information sources, and methods used in the design, implementation, and evaluation of *user interfaces*, the parts of software systems designed to interact with people. User interface *design* draws on the experiences of designers, current trends in input/output technology, cognitive psychology, human factors (ergonomics) research, guidelines and standards, and on the feedback from evaluating working systems. User interface *implementation* applies modern software development techniques to building user interfaces. User interface *evaluation* can be based on empirical evaluation of working systems or on the predictive evaluation of system design specifications.

Support Materials for User Interface Development

Edited by Gary Perlman, Ohio State University

SEI-SM-17-1.0

This support materials package includes materials helpful in teaching a course on user interface development.

Software Requirements

John Brackett, Boston University

SEI-CM-19-1.2

This curriculum module is concerned with the definition of software requirements—the software engineering process of determining what is to be produced—and the products generated in that definition. The process involves: (1) requirements identification, (2) requirements analysis, (3) requirements representation, (4) requirements communication, and (5) development of acceptance criteria and procedures. The outcome of requirements definition is a precursor of software design.

Formal Verification of Programs

Alfs Berztiss, University of Pittsburgh; Mark Ardis, SEI

SEI-CM-20-1.0

This module introduces formal verification of programs. It deals primarily with proofs of sequential programs, but also with consistency proofs for data types and deduction of particular behaviors of programs from their specifications. Two approaches are considered: verification after implementation that a program is consistent with its specification, and parallel development of a program and its specification. An assessment of formal verification is provided.

Software Project Management

James E. Tomayko, The Wichita State University; Harvey K. Hallman, SEI

SEI-CM-21-1.0

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.

Design Methods for Real-Time Systems

Hassan Gomaa, George Mason University

SEI-CM-22-1.0

This module describes the concepts and methods used in the software design of real-time systems. It outlines the characteristics of real-time systems, describes the role of software design in real-time system development, surveys and compares some software design methods for real-time systems, and outlines techniques for the verification and validation of real-time designs. For each design method treated, its emphasis, concepts on which it is based, steps used in its application, and an assessment of the method are provided.

Technical Writing for Software Engineers

Linda Levine, Linda Hutz Pesante, Susan Dunkle, Carnegie Mellon University

SEI-CM-23

ADA223872

This module, which is directed specifically to software engineers, discusses the writing process in the context of software engineering. Its focus is on the basic problem-solving activities that underlie effective writing, many of which are similar to those underlying software development. The module draws on related work in a number of disciplines, including rhetorical theory, discourse analysis, linguistics, and document design. It suggests techniques for becoming an effective writer and offers criteria for evaluating writing.

Concepts of Concurrent Programming

David Bustard, University of Ulster

SEI-CM-24

ADA223897

A concurrent program is one defining actions that may be performed simultaneously. This module discusses the nature of such programs and provides an overview of the means by which they may be constructed and executed. Emphasis is given to the terminology used in this field and the underlying concepts involved.

Language and System Support for Concurrent Programming

Michael B. Feldman,
George Washington University

SEI-CM-25

ADA223760

This curriculum module is concerned with support for concurrent programming provided to the application programmer by operating systems and programming languages. This includes system calls and language constructs for process creation, termination, synchronization, and communication, as well as nondeterministic language constructs such as the selective wait and timed call. Several readily available languages are discussed and compared; concurrent programming using system services of the UNIX operating system is introduced for the sake of comparison and contrast.

Support Materials for Language and System Support for Concurrent Programming

Edited by Gary Ford, SEI

SEI-SM-25

ADA223760

This package contains examples of concurrent programs written in Ada, Concurrent C, Co-Pascal, occam, and Modula-2. Machine-readable source code for the programs is available.

Understanding Program Dependencies

Norman Wilde,
University of West Florida

SEI-CM-26

A key to program understanding is unravelling the interrelationships of program components. This module discusses the different methods and tools that aid a programmer in answering the questions: “How does this system fit together?” and “If I change this component, what other components might be affected?”

Selected SEI Educational Support Materials

Teaching a Project-Intensive Introduction to Software Engineering

James E. Tomayko

CMU/SEI-87-TR-20

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. Appendices A-Z of this report contain materials used in teaching the course and the complete set of student-produced project documentation. These are available for $55.00 ($20.00 for the first copy sent to an Academic Affiliate institution).

Software Maintenance Exercises for a Software Engineering Project Course

Charles B. Engle, Jr., Gary Ford, Tim Korson

CMU/SEI-89-EM-1

This report provides an operational software system of 10,000 lines of Ada and several exercises based 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.)

The APSE Interactive Monitor: An Artifact for Software Engineering Education

Charles B. Engle, Jr., Gary Ford, James E. Tomayko

CMU/SEI-89-EM-2

In 1987 the SEI began a search for a well-documented Ada system, developed under government contract, that could be used in software engineering education. The APSE Interactive Monitor (AIM) was determined to be appropriate for this purpose. This system acts as an interface between a user of an Ada programming support environment (APSE) and the programs that the user executes in the APSE. It provides facilities to support the concurrent execution of multiple interactive programs, each of which has access to a virtual terminal. Educational uses of the system are described, including use as a case study and as the basis for exercises. Software engineering topics that can be taught with the system include software maintenance, configuration management, software documentation, cost estimation, and object-oriented design.

Program Reading: Instructor’s Guide and Exercises

Lionel Deimel, Fernando Naveda

CMU/SEI-90-EM-3

ADA228026

The ability to read and understand a computer program is a critical skill for the software developer, yet this skill is seldom developed in any systematic way in the education or training of software professionals. These materials discuss the importance of program reading, and review what is known about reading strategies and other factors affecting comprehension. These materials also include reading exercises for a modest Ada program and discuss how educators can structure additional exercises to enhance program reading skills.

Conference Proceedings

The conference and workshop records below are available directly from Springer-Verlag. Prices are indicated. Please send orders directly to the publisher: Book Order Fulfillment, Springer-Verlag New York, Inc., Service Center Secaucus, 44 Hartz Way, Secaucus, NJ 07094. Please specify the ISBN number when ordering.

Software Engineering Education: The Educational Needs of the Software Community

Norman E. Gibbs and Richard E. Fairley, editors

ISBN 0-387-96469-X

This volume contains the extended proceedings of the 1986 Software Engineering Education Workshop, held at the SEI and sponsored by the SEI and the Wang Institute of Graduate Studies. This workshop of invited software engineering educators focused on master's level education in software engineering, with some discussion of undergraduate and doctoral level issues. Hardback, $32.00.

Issues in Software Engineering Education: Proceedings of the 1987 SEI Conference

Richard Fairley and Peter Freeman, editors

ISBN 3-540-96840-7

Proceedings of the 1987 SEI Conference on Software Engineering Education, held in Monroeville, Pa. Hardback, $45.00.

Software Engineering Education: SEI Conference 1988

Gary Ford, editor

ISBN 3-540-96854-7

Proceedings of the 1988 SEI Conference on Software Engineering Education, held in Fairfax, Va. (Lecture Notes in Computer Science No. 327.) Paperback, $20.60.

Software Engineering Education: SEI Conference 1989

Norman E. Gibbs, editor

ISBN 3-540-97090-8

Proceedings of the 1989 SEI Conference on Software Engineering Education, held in Pittsburgh, Pa. (Lecture Notes in Computer Science No. 376.) Paperback.

Software Engineering Education: SEI Conference 1990

Lionel E. Deimel, editor

ISBN 3-540-97274-9

Proceedings of the 1990 SEI Conference on Software Engineering Education, held in Pittsburgh, Pa. (Lecture Notes in Computer Science No. 423.) Paperback.

Appendix 4. History and Acknowledgements

SEI curriculum development efforts began in the fall of 1985, when the staff of the SEI Graduate Curriculum Project developed a strawman description of the important subject areas and possible courses for a professional Master of Software Engineering (MSE) degree. This document was reviewed by the participants at the February 1986 SEI Software Engineering Education Workshop [Gibbs87], who offered numerous suggestions for improvement.

We then wrote a revised version of the document, which was widely circulated for additional comments. Those comments were analyzed over the winter of 1986-87, and in May 1987 the SEI published *Software Engineering Education: An Interim Report from the Software Engineering Institute* [Ford87]. This report was our first publication of curriculum recommendations, and it addressed not only curriculum content but also the related curriculum issues of educational objectives, prerequisites, student project work, electives, and resources needed to support the curriculum.

The interim report came to be regarded as a *specification* for an MSE curriculum because it concentrated on the content of the curriculum rather than how that content might be organized into courses or how those courses might be taught. We expected future work to include curriculum *design* (the organization of that content into meaningful courses), *implementation* (the detailed description of each course by instructors of the course), and *execution*, the process of teaching each course. (We have not yet planned a *validation* effort, though we see the need to do so.)

Two events in 1987 made it clear that a curriculum design was needed immediately. First, the SEI established a new project, the Video Dissemination Project, which would work with cooperating universities to offer graduate-level software engineering courses on videotape. Second, Carnegie Mellon University made a commitment to establish an MSE program within its School of Computer Science. Both of these efforts needed a curriculum, including detailed designs for courses.

In February 1988, the SEI sponsored the Curriculum Design Workshop, whose goal was to design an MSE curriculum that was consistent with the specification in the interim report. The workshop produced designs for six core courses. Those courses were described in our 1989 curriculum report [Ardis89].

The staff of the SEI Video Dissemination Project implemented and taught those six courses during the period 1988-1990. Those implementations are described in Chapter 3 of this report.

The curriculum recommendations in this report have benefitted from the efforts of many people. We had valuable discussions with many members of the SEI technical staff, including Mario Barbacci, Dan Berry, Barbara Callahan, Maribeth Carpenter, Clyde Chittister, Lionel Deimel, Larry Druffel, Peter Feiler, Robert Firth, Priscilla

Fowler, Dick Martin, Nancy Mead, John Nestor, Joe Newcomer, Mary Shaw, Nelson Weiderman, Chuck Weinstock, and Bill Wood, and with visiting staff members Bob Aiken, Alfs Berztiss, John Brackett, Brad Brown, David Budgen, Fred Cohen, Jim Collofello, Chuck Engle, Bob Glass, Paul Jorgensen, Nancy Leveson, Ev Mills, Larry Morell, Dieter Rombach, Rich Sincovec, Joe Turner, and Peggy Wright. We also acknowledge ideas from Harlan Mills and John Musa.

Earlier versions of the MSE recommendations were written by Jim Collofello and Jim Tomayko, and reviewed by Evans Adams, David Barnard, Dan Burton, Phil D'Angelo, David Gries, Ralph Johnson, David Lamb, Manny Lehman, John Manley, John McAlpin, Richard Nance, Roger Pressman, Dieter Rombach, George Rowland, Viswa Santhanam, Walt Scacchi, Roger Smeaton, Joe Touch, and K. C. Wong.

An early version of the MSE curriculum was the subject of discussion at the Software Engineering Education Workshop, which was held at the SEI in February 1986 [Gibbs87]. In addition to several of the people mentioned above, the following participants at the workshop contributed ideas to the current curriculum recommendations: Bruce Barnes, Victor Basili, Jon Bentley, Gordon Bradley, Fred Brooks, James Comer, Dick Fairley, Peter Freeman, Susan Gerhart, Nico Habermann, Bill McKeeman, Al Pietrasanta, Bill Richardson, Bill Riddle, Walter Seward, Ed Smith, Dick Thayer, David Wortman, and Bill Wulf.

The six MSE core courses were developed by Mark Ardis, Jim Collofello, Lionel Deimel, Dick Fairley, Gary Ford, Norm Gibbs, Bob Glass, Harvey Hallman, Tom Kraly, Jeff Lasky, Larry Morell, Tom Piatkowski, Scott Stevens, and Jim Tomayko.

The courses in the SEI Academic Series were developed by Mark Ardis, Dan Berry, Robert Firth, Harvey Hallman, and Jim Tomayko.

Bibliography

Amoroso88 — Amoroso, Serafino, Kuntz, Richard, Wheeler, Thomas, and Graff, Bud. “Revised Graduate Software Engineering Curriculum at Monmouth College.” *Software Engineering Education: SEI Conference 1988*, Ford, Gary A., ed. New York: Springer-Verlag, 1988, 70-80.

Ardis89 — Ardis, Mark, and Ford, Gary. *1989 SEI Report on Graduate Software Engineering Education*. Tech. Rep. CMU/SEI-89-TR-21, ADA219018, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., June 1989.

Bloom56 — Bloom, B. *Taxonomy of Educational Objectives: Handbook I: Cognitive Domain.* New York: David McKay, 1956.

Brackett88 — Brackett, John, Kincaid, Thomas, and Vidale, Richard. “The Software Engineering Graduate Program at the Boston University College of Engineering.” *Software Engineering Education: SEI Conference 1988*, Ford, Gary A., ed. New York: Springer-Verlag, 1988, 56-63.

Budgen86 — Budgen, David, Henderson, Peter, and Rattray, Chic. “Academic/Industrial Collaboration in a Postgraduate MSc Course in Software Engineering.” *Software Engineering Education: The Educational Needs of the Software Community*, Gibbs, Norman E., and Fairley, Richard E., eds. New York: Springer-Verlag, 1986, 201-211.

Collofello82 — Collofello, James S. “A Project-Unified Software Engineering Course Sequence.” *Proc. Thirteenth SIGCSE Technical Symposium on Computer Science Education*. New York: ACM, Feb. 1982, 13-19.

Comer86 — Comer, James R., and Rodjak, David J. “Adapting to Changing Needs: A New Perspective on Software Engineering Education at Texas Christian University.” *Software Engineering Education: The Educational Needs of the Software Community*, Gibbs, Norman E., and Fairley, Richard E., eds. New York: Springer-Verlag, 1986, 149-171.

Engle89 — Engle, Charles B., Jr., Ford, Gary, and Korson, Tim. *Software Maintenance Exercises for a Software Engineering Project Course*. Educational Materials CMU/SEI-89-EM-1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Feb. 1989. Includes distribution diskettes for software.

Ford87 — Ford, Gary, Gibbs, Norman, and Tomayko, James. *Software Engineering Education: An Interim Report from the Software Engineering Institute.* Tech. Rep. CMU/SEI-87-TR-8, ADA182003, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., May 1987.

Gibbs87	*Software Engineering Education: The Educational Needs of the Software Community*. Gibbs, Norman E., and Fairley, Richard E., eds. New York: Springer-Verlag, 1987.
Horning76	Horning, J. J. "The Software Project as a Serious Game." *Software Engineering Education: Needs and Objectives: Proceedings of an Interface Workshop*, Wasserman, Anthony, and Freeman, Peter, eds. New York: Springer-Verlag, 1976, 71-75.
IEEE83	*IEEE Standard Glossary of Software Engineering Terminology*. ANSI/IEEE Std 729-1983, IEEE, New York, 1983.
Lehman86	Lehman, Manny M. "The Software Engineering Undergraduate Degree at Imperial College, London." *Software Engineering Education: The Educational Needs of the Software Community*, Gibbs, Norman E., and Fairley, Richard E., eds. New York: Springer-Verlag, 1986, 172-181.
Mills86	Mills, Everald. "The Master of Software Engineering [MSE] Program At Seattle University After Six Years." *Software Engineering Education: The Educational Needs of the Software Community*, Gibbs, Norman E., and Fairley, Richard E., eds. New York: Springer-Verlag, 1986, 182-200.
Musa90	Musa, John D., Iannino, Anthony, and Okumoto, Kazuhira. *Software Reliability: Measurement, Prediction, Application*. New York: McGraw-Hill, 1990.
Norman88	Norman, Donald A. *The Psychology of Everyday Things*. New York: Basic Books, Inc. Publishers, 1988.
NRC85	National Research Council, Commission on Engineering and Technical Systems. *Engineering Education and Practice in the United States: Foundations of Our Techno-Economic Future*. Washington, D.C.: National Academy Press, 1985.
Scacchi86	Scacchi, Walter. "The Software Engineering Environment for the System Factory Project." *Proc. Nineteenth Hawaii Intl. Conf. Systems Sciences*. 1986, 822-831.
Wang86	*Bulletin, School of Information Technology 1986-1987*. Wang Institute of Graduate Studies, July 1986.
Webster83	*Webster's Ninth New Collegiate Dictionary*. Springfield, Mass.: Merriam-Webster Inc., 1983.
Wortman86	Wortman, David B. "Software Projects in an Academic Environment." *Software Engineering Education: The Educational Needs of the Software Community*, Gibbs, Norman E., and Fairley, Richard E., eds. New York: Springer-Verlag, 1986, 292-305.