Submitted In Partial Fulfillment of the Requirements for the Degree:

Doctor of Philosophy In

Software Engineering

With Areas of Specialization In
Project and Quality Management and Curriculum Development

The Union Institute Doctoral

Project Demonstrating Excellence

"Software Engineering Academic Project Management Production Tools (C-ProMPT)"

Core Faculty Advisor: Benjamin R. H. Davis, Ph.D.

Submitted By: Gregory E. Russell

Submitted To: The Graduate School of The Union Institute, Cincinnati, Ohio

February 10, 1996

Abstract

The Project Demonstrating Excellence consists of two parts. The first part includes a personal computer software application and accompanying on-line user guides. The second part is a contextual piece that contains a review of the relevant scholarly software engineering literature, software academic issues, and the development methodologies and issues involved with the software application development.

The software application was designed and developed to demonstrate a computer science and software engineering academic project management tool that is easy to use and provides consistent coding and document standards and principles. The system is called "Software Engineering Academic Project Management Production Tools" or "C-ProMPT." It includes the following modules: Personal Software Processes (process management) tools, forms, and guidelines; detailed C and C++ coding standards; academically proven software engineering document standards; document standard templates; document design guidelines; and other software engineering topics (software design, unit testing, risk management, and quality improvement). The system design incorporates object-oriented and event-oriented programming using a modified fourth-generation computer language and human-computer interaction principles.

The software is a Microsoft Windows application that interfaces completely with the Windows environment. Included in the application is extensive on-line context-sensitive help. A small users' guide is included to provide installation and tutorial instructions.

The contextual piece is organized into three sections. The first section reviews the relevant software engineering scholarly literature. This review includes a sub-section discussing the complexities involved with the software engineering field and another sub-section discussing the acceptance of the software engineering field in academia. The second section describes the software engineering methods used to develop the software application. The last section reflects on the student and software practitioner human/computer interaction review of the software application.

Table of Contents

1. Review of the Current Literature

Software Engineering As A Discipline

Introduction

This section begins with a discussion of the history of computer software development and the several definitions of Software Engineering. This is followed by a story of a software error that caused six deaths in the 1980s. This leads to a discussion on computer and software development, software development methods, complexity and risk. The last section discusses software process improvement and how this quality management practice is providing credibility to software developers and the software engineering field.

What is Software Engineering? Is Software Engineering a computer science specialty? Is a software engineer a super programmer?

The software engineering discipline evolved over the past two decades from a computer programming concept to a recognized engineering discipline.

There are many definitions of Software Engineering. Each definition produces a slightly different graduate software engineering program. Ian Sommerville defined software engineering as [Sommerville92]:

> ... a number of possible definitions of software engineering. Their common factors are that software engineering is concerned with software systems built by teams rather than by individuals, uses engineering principles in the development of these systems and includes both technical and non-technical aspects. As well as having a thorough knowledge of computing techniques, software engineers must be able to communicate orally and in writing. They should be aware of the importance of project management and should appreciate the problems system users may have in interacting with software whose workings they may not understand.

> Software does not simply mean the computer programs associated with some application or product. As well as programs, 'software' includes the documentation necessary to install, use, develop and maintain these programs. For large systems, the effort needed to write this documentation is often as great as the required for program development.

Tom Gilb said that a software engineer is [Gilb88]:

> ... not a programmer
>
> A design engineer, with software as a major discipline and probably at least one specialty discipline.
>
> The software engineer can translate cost and quality requirements into a set of solutions to reach the planned levels.
>
> Specialty examples: reliability engineer, maintainability, portability, human factors, quality control, general architecture.

Richard Fairley, another software engineering expert defined software engineering in this way [Fairley85]:

> Software engineering is the technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates.

Finally, Stephen Schach stated that [Schach93]:

> Software Engineering is a discipline whose aim is the production of quality software, delivered on time, within budget, and that satisfies the user's needs. In order to achieve this goal, a software engineer has to acquire a broad range of skills, both technical and managerial. These skills have to be applied not just to programming but to every phase of software production, from requirements to maintenance.

Another world renown expert, James Martin, created a term to describe another type of software engineering discipline, the information engineer. This expert comes from an information system background. The information engineer is someone who looks at the whole business situation (enterprise-wide) and determines how a specific project, and well as other projects, solve enterprise-wide problems. He defined information engineering as the [Martin89]:

> Information engineering is defined as the application of an interlocking set of formal techniques for the planning, analysis, design, and construction of information systems on an enterprise-wide basis or across a major sector of the enterprise.

James Martin also stated in the same text:

> Software engineering applies structured[, real-time, and object-oriented] techniques to one project. Information engineering applies structured[, real-time, and object-oriented] techniques to the enterprise as a whole, or to a large sector of the enterprise. The techniques of information engineering encompass those of software engineering in a modified form.
>
> Because an enterprise is so complex, planning, analysis, design, and construction cannot be achieved on an enterprise-wide basis without automated tools. Information engineering has been defined with reference to automated techniques as follows:
>
> - An Interlocking set of automated techniques in which enterprise models, data models, and process models are built up in a comprehensive knowledge base and are used to create and maintain data processing systems.
>
> Information engineering sometimes been described as
>
> - An organization-wide set of automated disciplines for getting the right information to the right people at the right time.

These four individuals correctly identified the role of the software engineer. James Martin may disagree, but I think the software engineer is extremely interested in joint projects, especially if this individual is a division software project manager dealing with several subcontractors working joint projects.

As a software engineer with many years of experience and almost three years of Ph.D. studies, I know that a softare engineer is more than the definitions provided by the above experts. The definition that fully defines software engineering is provided by Daniel M. Berry [Berry92]:

1. Software engineering is that form of engineering that applies:
    - a systematic, disciplined, quantifiable approach,
    - the principles of computer science, design, engineering, management, mathematics, psychology, sociology, and other disciplines as necessary,
    - and sometimes just plain invention,

    to creating, developing, operating, and maintaining cost-effective, reliably correct, high-quality solutions to software problems.
2. Software engineering is also the study of and search for approaches for carrying out the activities of (1) above.

In the interest of briefer sentences in the sequel, the phrase "quality software" means cost effective, reliably correct, high-quality solutions to software problems. Since cost effectiveness includes performance, "quality software" also means software that is performing adequately for its purpose. The word "producing" means creating, developing, operating, and maintaining; and "underlying principles" means principles of computer science, design, engineering, management, mathematics, psychology, sociology, and other disciplines as necessary.

The concept of software engineering was first envisioned almost thirty years ago. Dr. Stephen Schach discussed the first conference dedicated to software engineering [Schach93]:

> In the belief that software design, implementation, and maintenance could be put on the same footing as traditional engineering disciplines, a NATO study group in 1967 coined the term "software engineering." The claim that building software is similar to other engineering tasks was endorsed by the 1968 NATO Software Engineering Conference held in Garmisch, Germany. (This endorsement is not very surprising; the very name of the conference reflected the belief that software production should be an engineering-like activity.) A conclusion of the conferees was that software engineering should use the philosophies and paradigms of established engineering disciplines, and that this would solve what they termed the *software crisis*, namely, that the quality of software was generally unacceptably low and that deadlines and cost limits were not being met.

The software crisis term is refered to in many software engineering text books and articles. Dr. Roger Pressman described the term in this manner [Pressman92]:

> Many industry observers (including this author in earlier editions of this book) have characterized the problems associated with software development as a "crisis." Yet, what we really have may be something rather different.
>
> The word "crisis" is defined in Webster's Dictionary as "a turning point in the course of anything; decisive or crucial time, stage or event." Yet, for software there has been no "turning point," no "decisive time," only slow, evolutionary change. In the software industry, we have had a "crisis" that has been with us for close to 30 years, and that is a contradiction in terms.
>
> Anyone who looks up the word "crisis" in the dictionary will find another definition: "the turning point in the course of a disease, when it becomes clear whether the patient will live or die." This definition may give us a clue about the real nature of the problems that have plagued software development.
>
> We have yet to reach the stage of crisis in computer software. What we really have is a *chronic afflictions* (This terminology was suggested by Professor

Daniel Tiechrow of the University of Michigan in a talk presented in Geneva, Switzerland, April 1989). The word "affliction" is defined in Webster's as "anything causing pain or distress." But it is the definition of the adjective "chronic" that is the key to our argument: "lasting a long time or recurring often; continuing indefinitely." It is far more accurate to describe what we have endured for the past three decades as a chronic affliction rather than a crisis. There are no miracle cures, but there are many ways that we can reduce the pain as we strive to discover a cure.

Whether we call it a software crisis or a software affliction, the term alludes to a set of problems that are encountered in the development of computer software. The problems are not limited to software that "doesn't function properly." Rather, the affliction encompasses problems associated with how we develop software, how we maintain a growing volume of existing software, and how we can expect to keep pace with a growing demand for more software. Although reference to a crisis or even an affliction can be criticized for being melodramatic, the phrases do serve a useful purpose by encompassing real problems that are encountered in all areas of software development.

This afflication is not rare in the software development industry. Unfortunately, it is common. It is so common that the normal software manager and software practitioner accept the afflication as part of the business. In other words, these individuals do recognize the symptoms that may cause their software products to destroy businesses and human lifes.

Before I continue with the computer and software evolution discourse, it is imperative that the reader understand hazards associated with this afflication. The following story will help the reader understand the afflication better.

Benevolent Odious Software

While attending the Software Engineering Institute's 4th Annual Conference on Risk, I talked with one of the nation's renown experts on software risk, Dr. Peter Neumann, Stanford Research Institute. We talked about why organizations were having such a hard time accepting risk analysis and management principles in their software development practices. There were many reasons, but the predominate one was a misundertanding of risk itself. He said that companies perfer to manage problems, "putting out fires," rather than "fire prevention." People are promoted for putting out fires; those that practice fire prevention are not as visible. He also told me about an X-ray machine that killed six people because the manufacturer was more concerned about "putting out fires." This story appears in Ivars Peterson's book *Fatal Defect* [Peterson95].

> Everything appeared normal on the morning of March 21, 1986, when Ray Cox returned to the East Texas Cancer Center in Tyler to receive treatment for a tumor in his upper back. Several months earlier, doctors had removed a cancerous growth from this region of the thirty-three-year-old oil field worker's body, and the patient was now nearing the end of a regimen of therapeutic radiation treatments. Cox lay facedown on a table beneath the arm of a high-tech radiation therapy machine known as the Therac-25. Eight previous treatments had taught Cox that it was a painless procedure, no more disturbing than sitting for a photograph.
>
> This time, however, Cox experienced a sharp jolt, a sensation resembling a strong electrical shock. At the same instant it shot through his body, he heard an unfamiliar buzzing sound from the equipment. His back felt as if someone had accidentally spilled a cup of scalding hot coffee over it. Alone in the radiation-therapy room, he started to pull himself from the treatment table. But just as he was getting up, a second burst struck his arm. Cox later recalled that it felt as if his arm had suffered an electrical shock and that his hand were leaving his body. Seeking help, he tumbled off the table, staggered across the room, and began pounding on the door.

Outside the room's seven-foot-thick concrete walls. the technician operating the computer-controlled radiation machine hadn't seen Cox's reaction. On this particular day, the room's video monitor was disconnected and the intercom wasn't working. The only indication that anything might be amiss was a cryptic message — "malfunction 54" — that had appeared twice on the computer display outside the treatment room. Hearing the pounding, the operator immediately opened the door. She was shocked to find a shaken and injured Cox.

Cox was immediately taken to a nurse's station, where a physician examined him. Cox feared that he had suffered a radiation overdose, but the Therac-25's computer display suggested, if anything, that an underdose had occurred. Showing reddened skin in the treatment area but no obvious signs of serious injury, Cox was sent home. The clinic's staff checked the machine but failed to uncover any problems. The Therac-25 went back into service the same day and successfully completed its schedule of treatments.

That night, finding the pain in his neck and shoulder worsening, Cox checked into a hospital emergency room. A disfiguring mass had developed under the skin on his back, and doctors suspected that he had suffered an intense electrical jolt. When the cancer clinic was notified of this development, there was sufficient concern about a possible electrical or radiation problem that clinic personnel shut the machine down for testing. But they couldn't reproduce the conditions that had led to malfunction 54. According to the manufacturer's manual, this particular error message meant that the machine had delivered *either* an underdose or an overdose of radiation, but there was no clear evidence this had happened. After an independent engineering report vouched for the machine's electrical safety, the clinic returned its Therac-25 to service on April 7.

On April 11, malfunction 54 surfaced again at the Tyler clinic, with the same machine and the same technician. The victim was sixty-six-year-old bus driver

> Vernon Kidd, who was being treated for a skin cancer on the side of his face. This time the intercom was working, and the operator heard a loud noise and immediately rushed into the treatment room, where she found the patient moaning for help. Kidd had seen a brilliant flash of light, and he had heard an accompanying sizzling sound reminiscent of eggs frying. The side of his face felt as if it were on fire.
>
> Three weeks later, Kidd died. An autopsy revealed a high-dose radiation injury to the right lobe of his brain and brain stem. Meanwhile, Cox lost the use of his left arm and experienced periodic bouts of nausea and vomiting. He was eventually hospitalized for radiation-induced damage to his spinal chord, which caused paralysis of both legs and other complications. He died in September.

Mr. Peterson continues with a description of the faulty machine and the intial investigation:

> The machine at fault was the Therac-25 linear accelerator, a sophisticated, powerful device designed to fire a penetrating, high-energy beam of radiation deep into a patient's body to destroy embedded cancerous cells without injuring the surrounding tissue. From a port in its bulky, cantilevered arm, the Therac-25 could deliver radiation in two forms: either as a beam of electrons or as a beam of X rays. The accelerator produced the highly penetrating X rays by slamming a stream of high-energy electrons into a metal target. In the electron-producing mode, the machine would automatically move the metal target, decrease the electron beam's energy level, and send the beam directly to the tumor. Because a low-energy electron beam has less penetrating power than an X-ray beam, physicians use it to treat superficial cancers near the skin.
>
> The crucial element in the machine's design was a turntable that carried the devices which modified the electron beam for a particular form of radiation treatment into position. At the first setting, high-energy electrons struck a metal target to produce X rays; in the second position, scanning magnets

spread out the electron beam to a safe concentration for direct, electron-beam treatments; and in the third position, a mirror interrupted the electron beam path and a source of visible light illuminated the patient's body so the technician could aim it correctly. Three microswitches attached to the turntable monitored its position, and signals from these switches told the Therac-25's computer where the turntable was at any moment.

Initially the manufacture could not find the problem. Only after a clinic physicist at Taylor, Dr. Fritz Hager, discovered that an experienced operator could enter operational parameters faster than the machine could accept the entries. If the operator accidently entered a "X" for X-ray then corrected the mistake the machine would not detect the error or change. It took about six seconds for the machine to change from X-ray to electronic beam. During this time the operator's entry was not monitored.

An expert computer scienctist, Dr. Nancy G. Leveson, found many problems with the machine. According to Ivars Peterson [Peterson95]:

> The Therac-25's computer program, consisting of about twenty thousand instructions, had been written by a single programmer over a period of several years. It incorporated parts [software components] of the Therac-6 and Therac-20 programs, along with a great deal of new material tailored to the Therac-25's special features. Curiously, very little information about who this individual programmer emerged. The employee's employment records were not found, and the company employees, who filed depositions, could not provide information about this person's education, qualifications, or experience. It is known that this programmer left the company in 1986.
>
> Leveson and others who had a chance to examine the software were appalled by the mess they found. There was very little documentation — nothing written out to explain in plain English what different parts of the program did. There was no analysis demonstrating that key strings of instructions led to appropriately timed actions. There was no evidence that the software itself had been extensively tested before being bundled with the machine. The whole

package displayed shoddy, naive programming, but unfortunately, it typified the informal, undisciplined approach taken by many software developers in the 1970s.

Although Mr. Peterson's statement about software development in the 1970s is true, the implication that software developers have changed their software development methods in the 1980s and 1990s is not true. Software developers are still "hacking" code for mission-critical and life-critical software. This is the "software crisis" that so many software engineering experts are trying to overcome. Albeit, Dr. Pressman's statement is more to the point, this is a software *affliction.*

Why did this affliction occur? To the software user, the software, in most cases, is very simple to use. Most users feel, that if it is simple to use, it must be simple to develop. Software development is like a recent lava flow. At the surface it is nice and clean. You can walk on the surface if you are careful and don't mind the heat. However, below the surface the lava is still reeling and churning, trying desperately to break the surface shell. On the surface, software is nice and easy to use. Underneath the surface is a complex mixture of algorithms, logic, and heuristics that are barely in check. One missing bit could bring the entire surface crashing down upon the complex inner workings. It is this rapidly increasing complexity that is part of the software affliction.

Evolution of Computers

Computer development

The first computer systems and the programs that instructed the computers were very simple. Computer Science and computer scientist were unheard of. The computer science field was just starting to mature. As with any organism, this change from child to adult is very difficult. The computer science field is about sixty years old, and the software engineering field about thirty years old. As compared with other engineering fields these two fields are extremely new and in some cases, for example, construction engineering, these fields could be considered still in the embryonic stage. When did computers and software come into existence? According to Michael B. Feldman and Elliot B. Koffman the true digital computer age began [Feldman93]:

> ... In the late 1930s by Dr. John Atanasoff at Iowa State University. Atanasoff designed his computer to perform mathematical computations for graduate students.
>
> The first large-scale, general-purpose electronic digital computer, called the ENIAC (Electronic Numeric Integrator And Computer), was built in 1946 at the University of Pennsylvania. Its design was funded by the U.S. Army, and it was used to compute ballistics tables, predict the weather, and make atomic energy calculations. The ENIAC weighed 30 tons and occupied a 30-by-50-foot space.
>
> Although we are often led to believe otherwise, computers cannot reason as we do. Basically, computers are devices that perform computations at incredible speeds (more than one million operations per second) and with great accuracy. However, to accomplish anything useful, a computer must be programmed, that is, given a sequence of explicit instructions (a program) to perform.
>
> To program the ENIAC, engineers had to connect hundreds of wires and arrange thousands of switches in a certain way. In 1946 Dr. John von Neumann, of Princeton University, proposed the concept of a stored-program computer: a program stored in computer memory rather than set by wires and switches. Von Neumann knew programmers could easily change the contents of computer memory, so he reasoned that the stored-program concept would greatly simplify programming a computer. Von Neumann's design was a success and is the basis of the digital computer as we know it today.

Computers developed from 1939 to now evolved through four stages. Computer scientists often use the term “first generation” to refer to electronic computers that used vacuum tubes (1939 - 1958), The second generation began in 1958 with the changeover to transistors. The third generation began in 1964 with the introduction of integrated circuits. The fourth generation began in 1975 with the advent of large-scale integrated circuits. Since then, change has come so rapidly, between six to 18 months, that computer scientist don't seem to be counting generations anymore.

Programming Language Development

Initially mathematicians and then computer scientists developed most of the programming languages used from the 1940s to now. Programming languages have gone through a similar evolutionary track as computer hardware. Since 1939 various research groups, international committees and computer companies designed over a thousand different programming languages. Most of these languages have never been used outside the group which designed them; while others, once popular, have been replaced by newer languages.

The first generation languages were based on the structures of the computers of the early 1960s. These languages were machine oriented and had linear data structures.

The second generation languages incorporated hierarchically nested data structures, block structures, structured control features, built in types, syntactic structures. These languages provided the capability to develop very complex systems as compared with the first generation languages.

The third generation languages incorporated user defined data types, the ability to nest data structures to any depth, records and enumeration types, and efficient control structures, **case** or **switch** statement.

The fourth generation languages (4GL's) are languages that were developed as result of the dissatisfaction of business users with large conventional languages like COBOL.

Most 4GL's are produced for a particular computer or range of computers, and the distinction between a language and a package is not always clear. These packages include but are not all inclusive:

Report Program Generator (RPG)

This was probably the first 4GL and was produced in the 1960s in response to customer requests for a simple language for the generation of reports.

Application Generators

These 4GL's generate solutions for routine applications. Typical operations include data entry, ideally with full checking, and updating of files and databases. Examples are Borland's Delphi and Microsoft's Visual Basic.

Query Languages

These 4GL's are used with databases and allow the user to ask questions relating to several fields of the basic data records. More sophisticated languages in this category also allow the user to update the database. The most widely used language in this category is SQL.

Decision-Support Languages

The intention of the designers of 4GL's of this type was to help the user make informed and therefore better decisions. Such languages, for example ORACLE and INGRES, provide the user with facilities to build databases and to then perform statistical calculations, such as an analysis of trends on the data.

4GLs are currently the most fluid area in programming language design, with new languages springing up and others withering away through lack of users. What is not being produced are well-designed and thought-out languages carefully tailored to the needs of the customer and which take into account current and future computer hardware.

Time Line, Reader's Digest Version

To appreciate the rapid advances in digital computers and programming languages, Michael B. Feldman and Elliot B. Koffman provided this time line with some modifications [Feldman93],

Date	Event
2000 BC	The abacus is first used for computations.
1642 AD	Blaise Pascal creates a mechanical adding machine for tax computations. It is unreliable.
1670	Gottfried von Leibniz creates a more reliable adding machine that adds, subtracts, multiplies, divides, and calculates square roots.
1842	Charles Babbage designs an analytical engine to perform general calculations automatically. Ada Augusta (a.k.a. Lady Lovelace) is a programmer for this machine.
1890	Herman Hollerith designs a system to record census data. The information is stored as holes on cards, which are interpreted by machines with electrical sensors. Hollerith starts a company that will become IBM.
1939	John Atanasoff, with graduate student Clifford Berry, designs and builds the first electronic digital computer. His project was funded by a grant for $650.
1946	J. Presper Eckert and John Mauchly design and build the ENIAC computer. It uses 18,000 vacuum tubes and costs $500,000 to build.
1946	John von Neumann proposes that a program be stored in a computer in the same way that data are stored. His proposal (called "von Neumann architecture") is the basis of modern computers.
1951	Eckert and Mauchly build the first general-purpose commercial computer, the UNIVAC.
1957	An IBM team led by John Backus designs the first successful programming language, Fortran, for solving engineering and science problems.

Date	Event
1958	The first computer to use the transistor as a switching device, the IBM 7090, is introduced.
1958	Seymour Cray builds the first fully transistorized computer, the CDC 1604, for Control Data Corporation.
1958	ALGOL 58 programming language developed for solving business, engineering, and science problems
1960	The Department of Defense publishes the COBOL programming language specification
1964	The first computer using integrated circuits, the IBM 360, is announced.
1964	John Kemeny and Thoman Kurtz design and implement the BASIC programming language as a language for teaching programming languages
1964	An IBM team designs PL/1 programming language, for solving business, engineering, and science problems.
1965	The CTSS (Compatible Time-Sharing System) operating system is introduced. It allows several people to use a single computer simultaneously.
1969	Smalltalk developed by Alan Kay as a Ph.D. dissertation.
1971	Nicklaus Wirth designs the Pascal programming language as a language for teaching structured programming concepts.
1972	Dennis Ritchie, Bell Laboratories, designed and implemented the C programming language
1975	The first microcomputer, the Altair, is introduced.
1975	The first supercomputer, the Cray-1, is announced.
1976	Digital Equipment Corporation introduces its popular minicomputer, the VAX 11/780.
1977	Steve Wozniak and Steve Jobs found Apple Computer.

Date	Event
1978	Dan Bricklin and Bob Frankston develop the first electronic spreadsheet, called VisiCalc, for the Apple computer.
1980	Bjarne Stroustup, Bell Laboratories, developed C++ (C with Classes) on top of C to provide much of what Smalltalk pioneered.
1981	Microsoft Corporation introduces MS-DOS 1.0
1981	IBM introduces the IBM PC.
1982	Sun Microsystem introduces its first workstation, the Sun 100.
1983	The Department of Defense publishes the Ada programming language specification. This is the result of the most extensive and most expensive language design effort ever launched.
1983	Borland International introduces its first product Turbo Pascal.
1984	Apple introduces the Macintosh, the first widely available computer with a "user-friendly" graphical interface using icons, windows, and a mouse.
1984	Intel releases the 80286 microprocessor
1985	Microsoft Corporation introduces Windows.
1986	Intel introduces the 80386 microprocessor
1988	Intel introduces the 80486 microprocessor
1994	Intel introduces the Pentium microprocessor
1995	Microsoft Corporation introduces Windows 95

This time-line only shows when the initial event occured. It doesn't show the rapid improvements of the languages or computer systems. These rapid advances have driven the conception and birth of Software Engineering.

Evolution of Software Engineering Methods

Valdis Berzins [Berzins91] states:

> Why is Software Engineering important? Software engineering is important because
>
> 1. Software has a large and increasing effect on people's lives, and
> 2. Software has a large and increasing cost.
>
> Software is needed to enable computers to perform useful tasks. People's lives are being affected by software in increasingly critical ways as software is developed to automate many new tasks. Some of the areas being partially automated include financial services, communications systems, design and manufacturing operations, management information systems, control of power generation and distribution systems, medical services, air travel, space exploration, and weapons systems. Computers can perform tasks that are too complicated or too time consuming for people to do manually, and they can often do those tasks faster, at lower cost, and with greater reliability than people can. As software technology improves, the range of functions that can be usefully automated will continue to expand. However, computers are useful only if the software operates correctly and performs the functions needed by the people using the computers. Most computer system faults are due to design errors in the software rather than unpredictable behavior of the hardware. ... Developing reliable, useful, and flexible software systems is one of the great challenges facing software engineers today.

We have already read Dr. Schach's description of the first Software Engineering conference. He also made this comment [Schach93]:

> The fact that the software crisis is still with us, over 25 years later, should tell us two things. First, the software production process, while resembling traditional engineering in some respects, has its own unique properties and

problems. Second, the software crisis should rather be termed the software depression, in view of its long duration and poor prognosis.

It is certainly true that bridges collapse less frequently than operating systems. Why then cannot bridge-building techniques be used to build operating systems? What the NATO conferees overlooked is that bridges are as different from operating systems as ravens are from writing desks.

A major difference lies in the attitudes of the civil engineering community and the software engineering community to the act of collapsing. When a bridge collapses, as the Tacoma Narrows bridge did in 1940, this almost always means that the bridge has to be redesigned and rebuilt from scratch.

In contrast, when an operating system crashes it may simply be possible to reboot the system in the hope that the set of circumstances which caused the crash will not recur. This may be the only thing to do if, as is often the case, there is no evidence as to the cause of the crash. The damage caused by the crash will usually be minor: a database partially corrupted, a few files lost. Even when damage to the file system is considerable, by using back-up data the file system can often be restored to a state not too far removed from the state it was in just before the crash occurred.

Now consider a real-time system, that is, a system which has to be able to respond to inputs from the real world as fast as they occur.. For most real-time systems ... there is usually some element of fault tolerance built into the system to minimize the effects of a crash. That is to say, the system is designed in such a way that, if the system fails, an attempt is automatically made to recover from the failure.

The very concept of fault tolerance highlights a major difference between bridges and operating systems. ... Bridges are assumed to be perfectly engineered; operating systems are assumed to be imperfectly engineered. This

> fundamental difference is why software cannot be "engineered," in the classical sense of the word.
>
> It might be suggested that this difference is only temporary. ... The flaw in this argument is that hardware, and hence the associated operating system, is growing faster in complexity than we can handle it. In the 1960s, we had multiprogramming operating systems, virtual memory was a major complicating factor of operating systems of the 1970s, and now we are attempting to come to terms with multiprocessor and distributed (network) operating systems. Until we can handle the complexity caused by the interconnections of the various components of a software product such as an operating system, we cannot hope to understand it fully, and if we do not understand it, we cannot hope to engineer it. To make matters worse, complexity is growing too fast for us to hope to be able to master it.

During the 1970s several "software engineering" experts devised several methods to reduce the increasing software development complexity. These methods are called structured methods for software analysis and design.

Structured Methods

During the 1950s and 1960s software development was very simple. Time shared systems were none existent in the 1950s and just started to show up in the mid-1960s. Prior to time shared systems, the computer systems were essentially personal computers. Only one programmer could operate the system at any given time. A programmer would write a routine; run it on the computer to see if it worked; incorporate the routine if it worked or rework the routine if it didn't. This is similar to programming on our current personal computers. As time shared computer systems started to make their way into business, engineering, and military establishments, more and more programmers could program the computer at any given time. Along with the computer system's increased capacity, the creation of software become more complex. More and more software development efforts were failing due to being over budget and not on schedule. Something had to be done to reduce the complexity.

Lem Ejiogu [Ejiogu91] describes the "structured revolution" that occurred as a result of the increase software development complexity:

> The structured revolution was born out of common dissatisfaction with the then technical know-how of developing computing systems and the consequent disquieting problems of productivity. Essentially, there was general lack of formal principles for thinking, planning, designing, and testing software systems. These critical problems can be summed up as:
>
> - Poor managerial control
> - Poor verifiability
> - Poor modifiability
> - Poor adaptability
> - Poor maintainability
> - Poor testability
> - Poor reliability
> - Late deliverability
> - Skyrocketing costs of development and maintenance
> - Inadequate education for professionals
>
> These problems, indirectly at least, motivated the creation of the term software engineering. In one respect, they are an indictment of poor management; in another, they reveal the absence of adequate principles on the part of many practitioners who simply did not exercise well their powers of intellectual management of complexity. However, human manageability and education were not really to blame. The times were to blame. The computer revolution descended on us with an avalanche of problems beyond our known levels of

> functionality. But it is gratifying to note that man has rebounded with a good measure of resilience. Given the rapid deployment of computer technology by industry and science within so narrow a period of human history, it cannot be refuted that the revolution has been intelligently channeled to creative productivity.

This revolution created the first generation of software methodologies. These methods were generally developed between the late 1960s and mid-1970s. Only a hand-full of software development organizations were innovated enough to incorporate the methods into their software development practices.

Ed Yourdon [Yourdon93] provides an overview of the three structured methods, programming, analysis, and design. He described the first generation methods as:

> The evolution of system development methods has been gradual, with many people contributing to their improvement. The [first generation methods are identified]:
>
> > ... with the various Structured techniques' developed during the late 1960s and 1970s. Structured techniques break down a complex problem into smaller components, with well defined inter-relationships between the components. [These components include:]
> >
> > - Structured programming
> >   - Sequence, selection, iteration and avoiding 'GOTOs'
> >   - Modular design and structure charts
> >   - Programming style
> >   - Data structures
> > - Structured design
> >   - Successive refinement
> >   - Abstraction

    - Techniques based on the semantics of the structure chart
    - Data refinement techniques
- Structured Analysis
    - Data flow diagrams
    - Top-down functional decomposition
    - Avoiding technological bias
    - Information modeling

The total concept of structured methods, programming and software development methods, consisted of breaking down, decomposing, the problem steps into smaller steps until the steps cannot be broken down any further. The fundamental concepts of the first generation software engineering methods were not new; they were derived from many diverse sources, including engineering, hierarchy theory, Structured Programming, and even human psychology. [Page-Jones88]

In 1978 Tom DeMarco recommended several changes to the current analysis techniques used by the software industry. What is amazing about these recommendations is that they sound very similar to object-oriented techniques. This is what Tom DeMarco [DeMarco79] wrote in 1979:

> I suggest we need to make the following additions to our set of analysis phase goals:
>
> - Problems of size must be dealt with using an effective method of partitioning.
> - Graphics have to be used wherever possible.
> - We have to differentiate between logical and physical considerations, and allocate responsibility, based on this differentiation, between the analyst and the user.

- We have to build a logical system model so the user can gain familiarity with system characteristics before implementation.

At the very least, we require three types of new analysis phase tools:

- Something to help us partition our requirement and document that partitioning before specification. For this I propose we use a Data Flow Diagram.
- Some means of keeping track of and evaluating interfaces without becoming unduly physical. Whatever method we select, it has to be able to deal with an enormous flood of detail — the more we partition, the more interfaces we have to expect. For our interface tool I propose that we adapt a set of Data Dictionary conventions, tailored to the analysis phase.
- New tools to describe logic and policy, something better than narrative text. For this I propose three possibilities: Structured English, Decision Tables, and Decision Trees.

Now that we have laid all the groundwork, it is easy to give a working definition of Structured Analysis:

Structured Analysis is the use of these tools:

- Data Flow Diagrams
- Data Dictionary
- Structured English
- Decision Tables
- Decision Trees

to build a new kind of ... Document, the Structured Specification.

Although the building of the Structured Specification is the most important aspect of Structured Analysis, there are some minor extras:

- estimating heuristics
- methods to facilitate the transition from analysis to design
- aids for acceptance test generation
- walkthrough techniques

He went on to describe what Structured Analysis should not do, primarily any thing does not directly deal with the "problem set." If you carefully review Tom DeMarco's proposal you will find processes (functions), data (objects), and decision trees/table (states). What is interesting is that every method contains object, states, and functions (OSF). If you know how to utilize the OSF techniques for one method, it is fairly easy to convert it to another method [John White 94].

From the 1960s to the mid-1980s hundreds of computer system platforms came into being (generally PC platforms) along with software applications that increased the software development complexity even more. Roger Pressman [Pressman92] described the inherent complexity and software applications.

> It is somewhat difficult to develop meaningful generic categories for software applications. As software complexity grows, neat compartmentalization disappears. The following software areas indicate the breadth of potential applications:
>
> **System Software** — System software is a collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) process complex, but determinate, information structures. Other system applications (e.g., operating system components, drivers, telecommunications processors) process largely indeterminate data. In either case, the system software area is characterized by heavy interaction with computer hardware; heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing, and sophisticated process management; complex data structures; and multiple external interfaces.

Real-Time Software — Software that monitors/analyzes/controls real-world events as they occur is called real-time. Elements of real-time software include a data gathering component that collects and formats information from an external environment, an analysis component that transforms information as required by the application, a control/output component that responds to the external environment, and a monitoring component that coordinates all other components so that real-time response (typically ranging from 1 millisecond to 1 minute) can be maintained. It should be noted that the term "real-time" differs from "interactive" or "time-sharing." A realtime system must respond within strict time constraints. The response time of an interactive (or time-sharing) system can normally be exceeded without disastrous results.

Business Software — Business information processing is the largest single software application area. Discrete "systems" (e.g., payroll, accounts receivable/payable, inventory, etc.) have evolved into management information system (MIS) software that accesses one or more large databases containing business information. Applications in this area restructure existing data in a way that facilitates business operations or management decision-making. In addition to conventional data processing application, business software applications also encompass interactive computing (e.g., point-of-sale transaction processing).

Engineering and Scientific Software — Engineering and scientific software has been characterized by "number crunching" algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, new applications within the engineering/scientific area are moving away from conventional numerical algorithms. Computer-aided design (CAD), system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

Embedded Software — Intelligent products have become commonplace in nearly every consumer and industrial market. Embedded software resides in read-only memory and is used to control products and systems for the consumer and industrial markets. Embedded software can perform very limited and esoteric functions (e.g., keypad control for a microwave oven) or provide significant function and control capabilities (e.g., digital functions in an automobile such as fuel control, dashboard displays, braking systems, etc.).

Personal Computer Software — The personal computer software market has burgeoned over the past decade. Word processing, spreadsheets, computer graphics, entertainment, database management, personal and business financial applications, external network, or database access are only a few of hundreds of applications. In fact, personal computer software continues to represent some of the most innovative human-interface designs of all software.

Artificial Intelligence Software — Artificial intelligence (AI) software makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Currently, the most active AI area is expert systems, also called knowledge-based systems. However, other application areas for AI software are pattern recognition (image and voice), theorem proving, and game playing. In recent years, a new branch of AI software, called artificial neural networks , has evolved. A neural network simulates the structure of brain processes (the functions of the biological neuron) and may ultimately lead to a new class of software that can recognize complex patterns and learn from past "experience."

Most of these software applications have been around for at least 20 years. The only applications that are new, as compared to system, scientific and engineering, and business, are personal computer and artificial intelligence applications. Yes, compilers, operating systems, and business systems are advancing, but they based their current functionality on twenty year old code. The problem is that we are building our software applications upon loose soil. It is only by chance that most of these systems do not come tumbling down to our feet with disastrous results. Roger Pressman [Pressman92] commented on this dilemma:

> At the risk of sounding melodramatic, the software industry today is in a position that is quite similar to the steel industry of the 1950s and 1960s. Across companies large and small, we have an aging "software plant;" there are thousands of critical software-based applications that are in dramatic need of refurbishing:
>
> - Information system applications written 20 years ago that have undergone 40 generations of changes and are now virtually unmaintainable. Even the smallest modification can cause the entire system to fail.
> - Engineering applications that are used to produce critical design data, and yet, because of their age and state of repair, are not really understood. No one has detailed knowledge of the internal structure of their programs.
> - Embedded systems (used to control power plants, air traffic, and factories, among thousands of applications) that exhibit strange and sometimes unexplained behavior, but that cannot be taken out of service because there's nothing available to replace them.
>
> It will not be enough to "patch" what is broken and give these applications a modern look. Early components of the software plant require significant re-engineering, or they will not be competitive during the 1990s and beyond.

Roger Pressman may be a bit pessimistic, but he is not the only one concerned with the current state of our software industry. Part of these concerns drove methodologists in the late 1970s and 1980s to develop another software development method called object-oriented. Object-oriented programming originated from Simula67 and SmallTalk80, languages developed in the 1967 and 1980 respectively [Sebesta93].

Object-Oriented Methods

Object-oriented software development methods evolved out of the programming methods due to the problems with using structured methods to formulate classes (a special object) and methods (functions and states). There were other problems that Lem Ejiogu [Ejiogu91] discusses:

> Since 1986, a new philosophy about systems design has been called Object-Oriented Programming (OOP). The proponents argue that the conventional Software Life Cycle (SLC) have "chronic problems." However, no alternative model of the SLC has been proposed as a solution. Whether this is real, or the full potential of the SLC model has not yet been tapped, or the overwhelming need is to completely automate the design process (even when the principles of design are still unformalized), or a new terminology is being hatched, remains to be seen. The new methodology is at this time in its hypothetical cycle.
>
> Some disciples think that OOP or OOSE (Object-Oriented Software Engineering) is the beginning of a new revolution and hence the end of the structured revolution. This is erroneous. Although its principles are yet to be formalized beyond hypotheses, OOP is actually a rigorous formalistic extension of the structured revolution; its basis of philosophy is a subclass of that of the structured revolution. Logically, for any revolution to be unique, its thesis of foundation must be distinct and independent. With its core thesis on data abstraction, encapsulation and inheritance (a nice term for hierarchy; note: these are fundamental terms/concepts from structured programming), OOP may be seen as a concentrated effort to bring mathematical reasoning to software engineering. While CASE may be seen as revolution in tools, OOP may be seen as a revolution in optimal implementation of software systems — design, coding, and testing in particular. However, it is only hoped that the same disease that plagued structured programming — proliferation of

> methodologies and tools simultaneous with poor theoretical formalism — will not strangle OOP.

Lem Ejiogu alluded that the same problems that "plagued" the structured methods are now starting to "plague" the object-oriented methods. Over the past few years I have seen several methodologists working with either consulting firms or computer-aided software engineering (CASE) vendors to peddle their methods. These methodologists have not done an extensive research study on their methods [refer to page 104 for a detailed account of Software Engineering research methods]. Daniel Berry [Berry92] cautioned:

> ... There are many people selling software engineering snake oil and many charlatans who do no substantial work. These people are described as evangelists for their own methods, which they claim will solve all the world's problems. Probably they have consulting companies [or CASE vendors] that sell the method for megabucks, and they are interested only in advancing the fortunes of the company.

As with all new products the buyer should differentiate between the hype and actual functionality of the product. Again we are dealing with methods or applications that automate methods that are advertised as the "silver bullet" to solve all the software development complexity problems. James Martin, the gentleman who created Information Engineering, understands the motivation behind the methodologist and CASE vendors (He developed the Information Engineering Method and created a CASE company to sell the method). He also recognizes that software development organizations need to reorganize and streamline their operations. One way, is to use newer methods or to fully understand the current methods. James Martin [Martin93] stated:

> Most enterprises (except very small ones) need redesigning today in order to take advantage of new technology and networks, streamline procedures, eliminate redundancy and bureaucracy, and empower the employees to add more value. Business process redesign is the most important function of Information Systems departments. To redesign the value chains of an

enterprise, the chains need to be modeled. Object-oriented (OO) modeling is the best way to do this. The models should reflect the business policies and rules, and OO tools should allow these to be translated into operational systems as automatically as possible. When the business policies change (which happens constantly), the business systems should be regenerated quickly to reflect the change.

What are object-oriented (OO) models and techniques? They are similar to the structured models and techniques. Although OO analysis and design are still very new and most of the promising methods have connectivity problems, that is, it is very difficult to transverse from analysis to design.

Structured analysis and design methods use two techniques to translate from analysis to design. These techniques are called transaction and transformation analysis. These two techniques allow the designer to chose where to place the high-level component based on whether the data is transformed or some type of transaction is being perform on the data within a specific process. From there the designer can determine where to place the other processes based on where those processes are leveled (hierarchical order) in the data flow.

Currently there is a huge gap between OO analysis and design. There are two methodologists earnestly working on the problem, Rumbaugh and Booch. Both have stated that they will have a viable OO analysis and design method that will provide the connectivity within the next year. When they do overcome the problem it will revolutionize software analysis and design processes. James Martin [Martin93] confirms this:

Today's software is relatively trivial. To make computers into synergistic partners for humans, they need complex software. Software of the necessary complexity probably cannot be built using traditional structured techniques alone. In the mid-1980s, authorities of structured techniques claimed that building the proposed systems of 50 million lines of code was impossible. Our future requires software in which systems of 50 million lines of code will be commonplace. Object-oriented techniques with encapsulation, polymorphism, repository-based development. design automation, and code generators are essential for this.

The advantage of object-oriented programming is that:

> An object's procedures and data are hidden from the other parts of a program. This is called "encapsulation." An object's data can only be manipulated from inside the object. An object's procedures are called its "methods." An object's methods can be changed internally without affecting the other parts of a program. Each object is independent and can be used in many different systems without changing the it program code[, this is referred to as reusable code or objects].
>
> Object-oriented programming is based on the concepts of "class" and "inheritance," Classes are general categories of similar objects. A class is never used directly in a program. Object are. A class is used for creating objects which are "instances" of that class. Objects belonging to a certain class "inherit" all then structures and behaviors of that class. Each object can be modified by adding variables and behaviors unique to the object. ... New classes of objects can be created by choosing an existing class and specifying how the new class differs from the existing class. [Walsh94]

What is the aim of OO analysis (OOA) and design (OOD)? First, let's discuss the design process, and the analysis process will fall right into place. The primarily aim of OOD is to determine the objects in a product and then to design the product in terms of those objects. As mentioned before, there are a number of OOD methods in the software engineering field. Although they differ with regard to detail, almost every version of OOD consists of the following four steps:

1. Define the problem as concisely as possible
2. Develop an informal strategy a general sequence of steps for satisfying the requirements specification subject to the internal and external constraints.

3. Formalize the strategy:
    - Identify the objects and their attributes.
    - Identify operations to be applied to the objects.
    - If possible, identify classes
4. Proceed to detailed design and implementation

Steps 1 and 2 are performed during the analysis or specification software life cycle phase. OOD itself is a technique applicable only to the architectural design phase.

Steps 2 and 3 may be applied stepwise (decomposed) until the architectural design is satisfactory. For large products, this stepwise approach is all but mandatory. However, this is not unexpected; stepwise refinement is used throughout software engineering in order to reduce the complexity.

Also, remember that I said that OO methods are similar to structured methods. If you review the above methods with those described by Tom DeMarco in the last section you find that they are very similar. The big difference between OO methods and structured methods is how you look at the problem. Structured methods commence by considering the system's behavior or data separately; object-oriented methods combines them and regards them as integrated objects. It is this integration of the system data and behavior that allows the analyst and designer to reduce the complexity of the system by creating a system abstract model. This model then can be implemented in either an OO programming language or standard procedural language (although not as easy as the OO language). James Martin [Martin93] describes some object-oriented development methods but primarily object-oriented programming techniques:

> The world of object-oriented techniques ... [are] different. The designer thinks in terms of objects and their behavior, and code is generated. ... Most systems can be built without having to think about loops, branches, and program control structures. The system builder learns a different style of thinking. Events cause changes in the state of objects. Most of these state changes require small pieces of code, so coding is less error prone. Object

> types are built out of other object types. If an object type works well, the designer should treat it as a black box which he never looks inside (just as you never look inside your VCR). Software engineering then assumes more of the characteristics of hardware engineering.
>
> The problem with conventional [structured] programming is that it allows the programmer to do anything he wants. Data can assume any structure and processes can do anything to the data that the programmer desires. A branch instruction can link to far-away code and change variables. Any instruction in a computer can modify any location in the machine's memory. The number of path combinations exceeds any capability to test them all. The program becomes unpredictable and uncontrollable.
>
> In object-oriented programming, each object is restricted to sending requests to other objects. An object receiving a request checks its validity and executes a method. Most methods are relatively simple and, by themselves, relatively easy to test.

This "ease" of design, implementation, and testing will come about as the OO methods mature. However, the ramifications of OO methods in the software engineering field are tremendous, especially reducing the complexity in system development and maintenance. According to James Martin [Martin93]:

> To help deal with the complexities, structured programming came into use. It reduced the spaghetti in code, but programming was still based on the expected sequence of executing instructions. The attempt to design and debug programs by thinking through the order in which the computer does things ultimately leads to software that nobody can fully understand.
>
> One of the most urgent concerns in the computer industry today is the need to create software and corporate systems much faster and at lower cost. To put the ever-growing, power of computers to good use, we need software of much greater complexity.

> OO techniques make software engineering more like hardware engineering in that software is built from existing components, where possible. Just as a hardware designer does not change a microchip, a software designer does not tamper with the software chips.

James Martin is referring to the ultimate goal of OO methods, using software components or reusable code. Unfortunately the industry is just beginning to recognize the benefits of structured methods, after being promoted for over twenty years. Although there are similarities between the two methods, the problem solving methods are extremely different. It this difference that will cause problems in a software development organization if they do not have a firm foundation in structured methods and an excellent footing in viewing the world as objects. James Martin [Martin93] also commented on this subject:

> Introducing OO technology can present problems, and some OO projects have failed. Like any other software technology, OO is not a panacea.
>
> To use OO technology well, much careful training is needed. It takes time for computer professionals to think in terms of encapsulation, inheritance, and the diagrams of OO analysis and design. After an attempted switch to OO , traditional analysts may still tend to think in terms of structured decomposition, dataflow diagrams and conventional database usage. They often think in terms of data independence rather than class encapsulation. C++ and other nonpure OO tools allow developers to use non-OO constructs, and some of them regress to non-OO design and programming.
>
> Good use of inheritance and reusable classes requires cultural and organizational changes. The class library needs to be well managed. In most organizations, building up the library of classes needed to achieve a high level of reusability will take a long time.

> Currently, OO tools, although exciting, suffer from immaturity compared with the well-established, traditional CASE tools. Some traditional tools have been given a flavor of OO but still require non-OO techniques.
>
> When a traditional project gets in trouble, it can often be rescued with skilled people. When a project with new tools and techniques gets in trouble, the talent may not be available to rescue it.
>
> Successful introduction of OO technology needs both good education for every developer and managers who know what they are doing. The support staff needs to be established. The developers who build classes are often separate from the developers who use classes. OO has succeeded spectacularly with individual developers and small skilled teams. However, introducing it to a large group of traditional developers is more difficult. The biggest payoff in OO technology comes when its use is widespread which maximizes reusability and minimizes maintenance costs.

The proponents of OO claim that the biggest advantage to OO methods is reduction in complexity during software analysis, design, implementation, and testing phases. However, the proponents of structured methods made that same claim almost twenty years ago [DeMarco79].

Software Complexity and Risk

Software engineers have to be wary of claims that are not fully validated in the industry. We will always have to deal with complexity in our products. If the software engineers and managers fully understand the software complexities inherent in their development practices and within the product itself, we can then come to grips with the problem successfully and then start to manage the processes in a mature manner.

Unfortunately, the vast majority of software practitioners and managers not only have a limited understanding of software methods, they do not understand the complexities involved with software development. I have met many middle and upper managers who firmly believe that given a good software development tool they can hire entry-level programmers to replace senior-level programmers. These managers have an unrealistic view of software and its inherent complexity. According to Tom DeMarco [DeMarco82]:

> What is this thing called "complexity," and how does it affect software development? The nature and effects of complexity have been studied for years by systems people. but our industry has not even been able to settle on a definition. In a charming essay on complexity, Bill Curtis was driven to this one:
>
> > *Complexity is a not so-warm feeling in the tummy.*
>
> Perhaps, when you first began the business of software development, you were exhorted, as I was, to "Keep it simple, Stupid." The flattering implication of this saying is that we software people are all to intelligent for our own good, and that is the root cause of complexity, if we were dumber, we could write simpler software. But simple software, we now know, is never produced by the simple-minded. Taking something that is inherently complex and making simple, or even a bit simpler, is a great intellectual achievement. The causes of complexity are so profound, and the pursuit of simplicity so difficult, that Niklaus Wirth, the man whose very name is synonymous with elegant simplicity, was led to this wistful remark:
>
> > *You vow to make it simple at all cost. You accept complexity as your enemy. Then you build it, doing your best to control complexity ... and it comes out complex anyway.*
>
> We may never have a firm enough intellectual grasp of complexity to eliminate it from our work.

Capers Jones [Jones91] identified 20 complexities inherent in either software development or within the software application itself:

All of the complexity research on software to date has been based on new programs. There are no pragmatic studies on complexity when updating an existing system, although empirical evidence reveals that updates have perhaps 3 times the error potential of new code for equal volumes and that errors correlate with structure. When the major forms of complexity that affect software projects are considered, there are at least 20 of them. As of 1991, only a few of them have been measured objectively and numerically; the rest still await exploration. The 20 varieties of complexity include the following:

1. *Algorithmic complexity* (deals with spatial complexity and algorithmic volumes). ... The basic concept is the length and structure of algorithms intended to solve various computable problems. ... Examples of problems with high algorithmic complexity include radar tracking and target acquisition.

2. *Computational complexity* (deals with chronological complexity and run time lengths). ... The basic concern is the amount of computer time or the number of iterations required to solve a computational problem or execute an algorithm. ... Examples of problems with high computational complexity include long-range weather prediction and cryptographic analysis.

3. *Informational complexity* (deals with entities and relationships). This form of complexity has become significant with the rise of large database applications. ... Examples of problems with high informational complexity include airline reservation systems, integrated manufacturing systems, and large inventory management systems.

4. *Data complexity* (deals with numbers of entity attributes and relationships). This form of complexity, similar in concept to informational complexity, deals with the number of attributes that a single entity might have. For example, some of the attributes that

might be used to describe a human being include sex, weight, height, date of birth, occupation, and marital status.

5. *Structural complexity* (deals with patterns and connections). This form of complexity deals with the overall nature of structures.
6. *Logical complexity* (deals with combinations of AND/OR/NOR/ NAND logic). This form of complexity deals with the kinds of logical operations that comprise syllogisms, statements, and assertions. It is much older than software engineering, but it has become relevant to software because there is a need for precise specification of software functions.
7. *Combinatorial complexity* (deals with permutations and combinations). This form of complexity deals with the numbers of subsets and sets that can be assembled out of component parts.
8. *Cyclomatic complexity* (deals with nodes and edges of graphs). Its basic concern is with the graph formed by the control flow of an application. Unlike some of the other forms of complexity, this one can be quantified precisely.
9. *Essential complexity* (deals with nodes and edges of reduced graphs). This form of complexity is similar in concept to cyclomatic complexity, but it deals with a graph after the graph has been simplified by the removal of redundant paths.
10. *Topologic complexity* (deals with rotations and folding patterns). This form of complexity is explored widely by mathematicians but seldom by software engineers. The idea is relevant to software, since it can be applied to one of the intractable problems of software engineering: attempting to find the optimal structure for a large system.
11. *Harmonic complexity* (deals with waveforms and Fourier transformations). This form of complexity is concerned with the

various waveforms that together create an integrated wave pattern. The topic is very important in physics and engineering, but it is only just being explored by software engineers.

12. *Syntactic complexity* (deals with grammatical structures of descriptions). This form of complexity deals with the structure and grammar of text passages. Although the field is more than 100 years old and is quite useful for software, it has seldom been utilized by software engineers. Its primary utility would be in looking at the observed complexity of specifications with a view to simplifying them for easier comprehension. It has a number of fairly precise quantifications, such as the FOG index and the Fleish index.

13. *Semantic complexity* (deals with ambiguities and definitions of terms). This form of complexity is often a companion to syntactic complexity. It deals with the definitions of terms and the meaning of words and phrases. Unlike syntactic complexity, it is rather amorphous in its results.

14. *Mnemonic complexity* (deals with factors affecting memorization). This form of complexity deals with the factors that cause topics to be easy or difficult to memorize.

15. *Perceptional complexity* (deals with surfaces and edges). This form of complexity deals with the visual appearance of artifacts and whether they appear complex or simple to the human perceiver. Regular patterns, for example, tend to appear simpler than random configurations with the same number of elements.

16. *Flow complexity* (deals with channels and fluid dynamics of processes). This form of complexity concerns fluid dynamics, and it is a major topic of physics, medicine, and hydrology. An entirely new subdiscipline of mathematical physics termed "chaos" has started to

emerge, and it seems to have many interactions with software engineering.

17. *Entropic complexity* (deals with decay and disorder rates). All known systems have a tendency to move toward disorder over time, which is equivalent to saying that things decay. Software, it has been discovered, also decays with the passage of time even though it is not a physical system. Each time a change is made, the structure of a software system tends to degrade slightly. With the passage of enough time, the disorder accumulates sufficiently to make the system unmaintainable.

18. *Functional complexity* (deals with patterns and sequences of user tasks). This form of complexity concerns the user perception of the way functions within a software system are located, turned on, utilized for some purpose, modified if necessary, and turned off again.

19. *Organizational complexity* (deals with hierarchies and matrices of groups). This form of complexity deals not with a software project directly, but with the organizational structures of the staff that will design and develop it. It has been studied by management scientists and psychologists for more than 100 years, but only recently has it been discovered to be relevant to software projects. A surprising finding has been that large systems tend to be decomposed into components that match the organizational structures of the developing enterprise rather than components that match the needs of the software itself.

20. *Diagnostic complexity* (deals with factors affecting identification of malfunctions). When a medical doctor is diagnosing a patient, certain combinations of temperature, blood pressure, pulse rates, and other signs are the clues needed to diagnose specific illnesses. Similarly, when software malfunctions, certain combinations of symptoms can be

used to identify the underlying cause. This form of complexity analysis is just starting to be significant for software projects.

Besides these complexities there are others. As software engineering researchers delve deeper into the "what" and "how" of software development more complexities will be discovered. Besides the 20 complexities identified by Tom DeMarco, Lem Ejiogu [Ejiogu91] identified three more complexities:

Psychological Complexity — expresses a measure of functional "fear" imposed on us (programmers, analysts, etc.) by a software project/task. The relative dwarfing effect of a project depends on our ability to fully comprehend its dimensions of configurations. This is why this behavior of software is derived from structural complexity. But it must be cautioned that conventional measures of programmer competence or speed of production are poor models of psychological complexity—that which is imposed on us must be distinguished from that which is a reflection (consequence) of our reaction. Comprehension or perception precedes performance; the two are independent events.

Cost Complexity — deals with the various ramifications of determining the cost of computing resources (including human and material) and forecasting in software productivity (allocation and scheduling). Current models of cost estimation rely on [counting] lines-of-code (LOC), but considering the so many lines of code deleted or modified during development or the undefined definition of what constitutes a LOC, these models are clearly uninformative, unscientific, and, therefore, defective. The cost of a product must effectively reflect the individual costs of its disparate components.

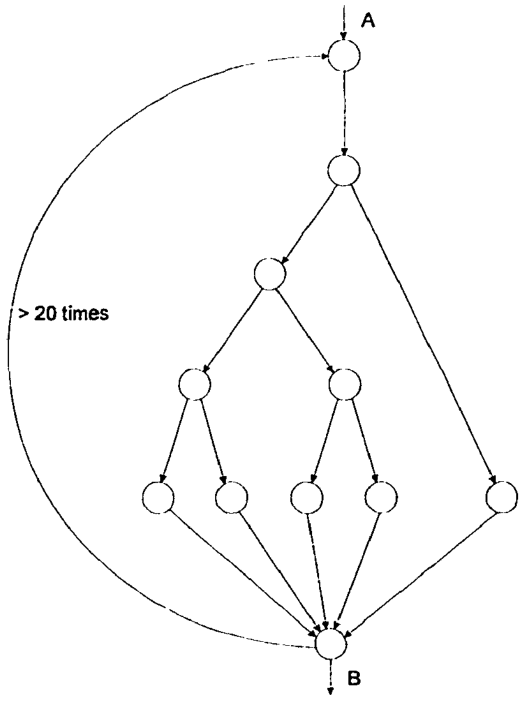
Readability Complexity — A post-code behavior that measures the degree of comprehension of a software module. Although related to psychological complexity, the characteristics here actually relate to complete and efficient refinement — each node carries a single thought.

Lem Ejiogu [Ejiogu91] futher commented on how software developers should be looking at the complexity classifications rather than the general term of "complex",

> This [complexity] classification can help do away with the myths that software complexity does not depend on (software) size; or that the number of bugs depends on program size (note: measurement of bugs is a quality issue, while that of size is a complexity issue); or that quality is the inverse of complexity. Henceforth, the word "complexity" will be understood as X-complexity where X is a category of complexity.

Software engineering researchers are just beginning to understand software complexity issues and how to measure them. To bring complexity "under control" we must measure the effects of complexity. Thomas McCabe developed a complexity measure during the mid-1980s. This measurement method is commonly termed the "McCabe complexity measures" or "cyclomatic complexity." Capers Jones [Jones86] discussed McCabe's method in some detail:

> McCabe's complexity measurement procedures is to graph the flow of control of a program as shown in this figure.



The McCabe technique is to count the number of regions in the resulting graph, where "regions" are defined as the surrounding outside area of the graph and all enclosed or bounded domains [(you count the nodes)]. Thus in the above figure, there are 11 regions in all, and hence the McCabe complexity measure would be 11.

McCabe has noted certain correlations between the complexity number and the real-life subjective difficulty of a piece of software:

- Modules or programs with a complexity number of less than 5 are usually considered simple.
- Modules or programs with a complexity number of greater than 5, but less than 10 are usually considered well structured and stable.
- In modules or programs with a complexity number of 20 or higher there appears to be a direct correlation between the number and subjective complexity.
- Modules or programs scoring higher than 50 are often error-prone and viewed as extremely troublesome.

In practical terms, the McCabe complexity metrics predict that as the number of branches in a program goes up, the complexity also goes up, and by implication, the number of bugs and errors should go up also.

The McCabe complexity measure has been one of the most successful trouble indicators or "bug predictors" yet discovered.

However, in spite of the success of McCabe's complexity measure, it only covers code-branching situations in finished programs and has no direct way of dealing with the complexity of the original problem.

The McCabe's complexity measure will at least help the software practitioners to understand the software product internals. We still need complexity measures for the requirements elicitation and the software development organizational structure. The previous discussions have pointed out that building software is only one aspect of a greater picture. The organizational structure contributes to software complexity as well. As software products increase in size, the communication paths increase exponentially. According to Lawrence Putnam [Putnam92] large-scale software development is extremely complex, just in the organizational nature.

> In the case of large systems, all of this complexity is spread over many people in different specialties. These people may be supervised by several layers of management and serviced by a variety of staff groups. Each person must communicate somehow to specific other persons what they need to know about his or her work and must receive, probably from still other people, what he or she needs to know all this without burdening everybody with everything.
>
> The complexity "makes overview hard, thus impeding conceptual integrity," Brooks went on. "It makes it hard to find and control all the loose ends. It creates the tremendous learning and understanding burden that makes personnel turnover a disaster."
>
> Those who have worked on large software projects can fill in the details of this spider's web themselves. For those who haven't, consider the detective novel. This type of novel runs about 100,000 words, or perhaps 10,000 short sentences. A high-level computer instruction is roughly equivalent to a short sentence in thought content. A large program of one million source lines of code would thus be equivalent to some 100 novels.
>
> Imagine trying to keep track of the plots, characters, weapons, and the milieus in 100 murder mysteries. For the software comparison moreover, all these novels would have to be tied together in one vast novel. All one hundred authors would have to be coordinated to write around one intricate plot. That planning and writing organization would take many layers of management on top of the prima donna authors. That is roughly the organizational situation in which much large-scale software development finds itself.

The results of all this complexity have been

- Poor planning decisions
- Cost overruns
- Schedule slippages
- Poor quality products
- Reduced-function products
- Unhappy users or customers
- Animosity within the organization.

There are at least three software engineering research institutions (Software Engineering Institute (SEI), Carnegie Mellon University, Software Engineering Laboratory, University of Maryland, and Software Productivity Research, Inc.) that are actively researching and validating methods to overcome the complexities previously discussed. However, for some reason software development organizations, managers and practitioners alike, disregard this work as "academic exercises" that do not have any validity for their organization. In other words, these organizations believe in the axiom, "Not invented here." Almost every organization that I have served as a software engineering consultant had this attitude. Lem Ejiogu [Ejiogu91] also commented on this problem.

> The phrase, "Intellectual Management of Complexity," attributed to Dr. Edsger W. Dijkstra, has come to be synonymous with programming
>
> *"I now suggest that we confine ourselves to the design and implementation of intellectually manageable programs"*
>
> It is an attempt to signal to the computing industry the real nature of the monster with which it has engaged as if in a battle. Contrary to misconceived and commonplace beliefs current in the days of ad hoc management of software productions, this phrase was intended to bring to the general

consciousness of practitioners the need for better education and, therefore, better understanding of the problems of software productivity.

The key words, "intellectual" and "complexity," serve exactly this purpose. In some environments, it was thought that software development (computer programming) required no more than a six-month crash course to be continued with some on-the-job exposure to coding, testing, designing, etc., in some programming languages. There are still practicing professionals today who feel that academic discourses and theoretical investigations of computer science are too far beyond what the industry needs, which they think is just a team of code manufacturers. Even some computer science departments continue to drill their students in the algorithms, programming languages, and "laboratory" exercises on coding and testing, with little or no emphasis on necessary mathematical disciplines, such as queuing theory, matrix theory, mathematical statistics, abstract algebra, topology, measure theory, and theory of measurement, etc. Essentially, theory is divorced nearly completely from applications as unrelated. What is worse, there are some educators and industry leaders and experts who decry the exploration of principled and theoretical ideas as delayed or wasteful productivity . They advocate immediate coding. These subjects were merely thought to benefit only students of programming languages, computer hardware designs, and operating systems.

Management and the software practitioners must take responsibility upon themselves to overcome any dificiencies they may have in software production methods and practices. As Ejiogu referred to above, we will not overcome most of our difficulties until we change the way we teach computer science and software engineering in our academic institutions and provide organizational training for software managers and practitioners as to the realities of software complexity and the methods to overcome them. Over the last 15 years the software industry has been trying to make these changes in our academic institutions with very little success [Christiansen92c].

Software Development Management

As academia must change the way it teaches computer science and software engineering programs, software project managers must change the way they manage software projects. We discussed the complexities inherent in software development and within the software itself. All of these complexities have to be managed by individuals with a keen insight on how to restrain them. Software project management comes in three flavors: project management, quality management, and process management. I call the combination of all the management skills, Software Development Management. A good software project manager is proficient in all three. Unfortunately, managers with these skills are few in number.

Within the next few pages I will try to highlight the problems that software development managers are facing now and why those problems exist. Grady Booch [Booch91] explained the main reason why we have tremendous problems with software development. It all started in the beginning ... :

> A physician, a civil engineer, and a [software engineer] were arguing about what was the oldest profession in the world. The physician remarked, "Well, in the Bible, it says that God created Eve from a rib taken out of Adam. This clearly required surgery, and so I can rightly claim that mine is the oldest profession in the world." The civil engineer interrupted, and said, "But even earlier in the book of Genesis, it states that God created the order of the heavens and the earth from out of the chaos. This was the first and certainly the most spectacular application of civil engineering. Therefore, fair doctor, you are wrong; mine is the oldest profession in the world." The [software engineer] leaned back in her chair, smiled, and then said confidently, "Ah, but who do you think created the chaos?"

Some of us believe that the Chaos theory was created to help the software practitioners and managers to better understand their environment. Out of chaos comes order and software development management is the attractor that will bring about that order. I have discussed the software environmental problems in part, but I have not explained the development phases that are encountered in every software development methodology, regardless of application area, project size or complexity. They are definition, development, and maintenance. According to Roger Pressman [Pressman92] the are defined as:

> The definition phase focuses on what. That is, during definition, the software developer attempts to identify what information is to be processed, what function and performance are desired, what interfaces are to be established, what design constraints exist, and what validation criteria are required to define a successful system. The key requirements of the system and the software are identified. Although the methods applied during the definition phase will vary depending upon the software engineering paradigm (or combination of paradigms) that is applied, three specific steps will occur in some form:
>
> *Systems analysis.* System analysis defines the role of each element in a computer-based system, ultimately allocating the role that software will play.
>
> *Software project planning.* Once the scope of the software is established, risks are analyzed, resources are allocated, costs are estimated, and work tasks and schedule are defined.
>
> *Requirements analysis.* The scope defined for the software provides direction, but a more detailed definition of the information domain and function of the software is necessary before work can begin.
>
> The development phase focuses on how. That is, during definition the software developer attempts to define how data structure and software architecture are to be designed, how procedural details are to be implemented, how the design will be translated into a programming language (or nonprocedural language), and how testing will be performed. The methods

applied during the development phase will vary, but three specific steps will always occur in some form:

Software design. Design translates the requirements for the software into a set of representations (some graphical, others tabular or language based) that describe data structure, architecture, algorithmic procedure, and interface characteristics.

Coding. Design representations must be translated into an artificial language (the language may be a conventional programming language or a nonprocedural language used in the context of the 4GL paradigm) that results in instructions that can be executed by the computer. The coding step performs this translation.

Software testing. Once the software is implemented in machine executable form, it must be tested to uncover defects in function, in logic, and in implementation.

The maintenance phase focuses on change that is associated with error correction, adaptations required as the software's environment evolves, and enhancements brought about by changing customer requirements. The maintenance phase reapplies the steps of the definition and development phases, but does so in the context of existing software. Three types of change are encountered during the maintenance phase:

Correction. Even with the best quality assurance activities, it is likely that the customer will uncover defects in the software. Corrective maintenance changes the software to correct defects.

Adaptation. Over time, the original environment (e.g., CPU, operating system, peripherals) for which the software was developed is likely to change. Adaptive maintenance results in modification to the software to accommodate changes to its external environment.

> *Enhancement.* As software is used, the customer/user will recognize additional functions that will provide benefit. Perfective maintenance extends the software beyond its original function requirements.

In extremely simple terms, these generic phases are performed by acquiring resources (material and personnel) and funds (cost). The final variables are the software functionality and the time provided to accomplish the task. James Lewis [Lewis95] described this relationship:

> For many years it has been customary to say that project management is the planning, scheduling, and controlling of project activities to achieve performance, cost, and time objectives, for a given scope of work, while using resources efficiently and effectively. These have been referred to as PCT objectives. They are also commonly called good, fast, and cheap. These more colorful terms capture the essence of what a project manager must achieve.
>
> The last sentence of the definition is really loaded! The three objectives must be met while using resources efficiently and effectively. This is a key point in project management, and one that is too often overlooked. Every organization has limited resources, and unless the project manager can deal successfully with the resource allocation problem, she will not be successful. Experience shows that in many environments failure to manage resources properly is one of the most common causes of project failure.
>
> The relationship among the four variables is given by the following equation:
>
> $$C = f(P,T,S)$$
>
> In words, the equation says, "Cost is a function of Performance, Time, and Scope." Ideally, a real equation could be written prescribing the actual relationships precisely. In practice, we never know that precise relationship. We have to estimate times and costs.

I have been using an equation similar to Lewis's equation. This equation looks at cost (C), time (T), resource (R), and functionality (F). While Lewis's equation deals with performance as a separate entity, I include this entity within functionality. The equation then becomes:

$$C = f(T,R,F)$$

This equation allows the manager to look specifically at the issues at hand, project cost, project duration or schedule, resources available, and implementation issues. With both equations you can manipulate three variables.

The problem with both of these equations is that you have to have a pretty good grasp of the cost estimates and system requirements. As I have discussed earlier, this is not generally the case in the initial project stage. So what good is the equation. I primarily use it to demonstrate to management that if a decision is made to reduce the schedule or cost or to increase the functionality, one or more of the other variables have to change. For example, if the schedule is shorten in duration you will have to decrease functionality or increase cost and resources. Wait a minute, you say? Why would you increase cost or resources? If you decrease the schedule time, why won't you decrease your effort (cost and resources)? This will only decrease, if you decease the functionality. If you have a 12 month project and reduce it by 10% and maintain the same functionality your total effort will be 52% greater than your 12 month effort [Humphrey95]. Conversely, if you increase your schedule by 10% you total effort will be reduced by 37% [Humphrey95]. These percentage are very simple to determine if you know how to use Taylor series methods.

Robert Block [Block83] also commented on this subject and discussed the common failures related to time, functionality (scope), and people (cost and resources),

> Resource failures involve conflicts of people, time, and scope; that is, the people and the amount of time allotted are not sufficient to build the required system. These failures are most often due to imposed deadlines combined with an inability or unwillingness by management to provide adequate resources. Resource failures result in systems that are late and frequently over budget.

Incorrect, incomplete, or unclear specification of system requirements leads to *requirement failures.*

Goal failures result from inadequate or incorrect statements of the system goals by management, or from a misunderstanding of the goals by the system builders.

Technique failures are failures by the system builders to use, or to use correctly, effective software development disciplines such as structured [or object-oriented] analysis and design.

User contact failures are caused by an inability to communicate with the user community.

Organizational failures result from an inability of the organizational structure to support the system building process.

Technology failures are failures of acquired hardware or software utilized by the system.

Although *size failures* can almost always be attributed to several of the other categories, the root of the problem is that the system is too big. Big systems are usually functionally complex and tend to push the system development capabilities of an organization to or beyond its limits.

Failures to motivate workers and to maintain the morale of the system building group are *people management failures.* The resulting lack of effort. stifled creativity, and antagonistic attitudes have an impact similar to that of internal organizational failures; only in this case, the fault lies not with the organization but with the group leader.

Methodology failures are failures to perform the activities needed to build the system: Unnecessary activities may be performed, needed activities may be omitted, or activities may be performed incorrectly. Methodology failures may be due to the lack of a formal methodology as a guideline to the system builders, or to an overly rigid adherence to the adopted methodology.

> *Planning and control failures* encompass planning, scheduling, task assignment, and tracking of results. Included here are vaguely defined assignments, inadequate tools to depict plans and schedules, and failure to track progress to insure that tasks are done.
>
> *Personality failures* are clashes between people either within one system building group or between group members (often the leader) and members of an interfacing organization; the failure results from people disliking each other enough to prevent them from doing their jobs. In the extreme, acts of sabotage and vengeance may occur, but more often there is passive cooperation and covert resistance.
>
> The impact of these failures varies with the functions and assignments of the individuals involved. They are rarely catastrophic, but often aggravate already difficult situations.

Lawrence Putnam [Putnam92] alluded to these failures when he described software managers who are not familiar with software development complexities:

> ... Software development process is hard to understand. It is particularly hard for managers who are divorced from software technology. They can't picture it, as they can a physical product in a drawing or prototype. To make matters worse, their staffs are often not literate in the subject either.
>
> Given the time restrictions under which top-level managers and their staffs function, it will not be simple to provide what they need to know about the software development process — not to design or program — but to function effectively at their own level.

Putnam's observation is not a rarity in the software industry; this is a common occurrence. Most of the companies I have consulted with have two type of managers, those with MBAs and those that have been successful in putting out fires. The first usually doesn't know a thing about software development; the second is usually a hacker who doesn't care about utilizing proven software development methodologies. Roger Pressman [Pressman92] observed the following about software managers:

> Middle- and upper-level managers with no background in software are often given responsibility for software development. There is an old management axiom that states: "A good manager can manage any project." We should add: ". . . if he or she is willing to learn the milestones that can be used to measure progress, apply effective methods of control, disregard [software development] mythology, and become conversant in a rapidly changing technology." The manager must communicate with all constituencies involved with software development customers, software developers, support staff, and others. Communication can break down because the special characteristics of software and the problems associated with its development are misunderstood. When this occurs, the problems associated with the software crisis are exacerbated.
>
> Software practitioners (the past generation has been called programmers; this generation is earning the title software engineer) have had little formal training in new techniques for software development. In some organizations a mild form of anarchy still reigns. Each individual approaches the task of "writing programs" with experience derived from past efforts. Some people develop an orderly and efficient approach to software development by trial and error, but many others develop bad habits that result in poor software quality and maintainability.

One of the major problems with software project management is estimating cost and resources. This is due to three reasons, incomplete requirement elicitation, poor software estimating methods, and complexity issues (which can be referred to as risks). The first can be dealt with by effectively communicating with the customer or client (although I don't anticipate the problem disappearing altogether). The second reason is slowly being resolved by software engineering researchers. Software project managers' incomplete understanding of risks and poor estimating techniques are reflected in their planned software project schedules. Initial project schedules are usually a ball-park picture of the manager's conception of the entire project based on incomplete data. Ian Sommerville [Sommerville92] described project scheduling as:

> One of the most difficult tasks of software management. Typically, projects break new ground. Unless the project being scheduled is similar to a previous project, previous estimates are not a good basis for new project scheduling. Different projects use different programming languages and methodologies, which complicates the task of schedule estimation.
>
> If the project is technically advanced, initial estimates will almost certainly be optimistic in spite of endeavors to consider all eventualities. In this respect, software scheduling is no different from scheduling any other type of large advanced project. New aircraft, bridges and even cars are frequently late because of unanticipated problems. Schedules, therefore, must be continually updated as better progress information becomes available.

There are several models to help the software project managers derive reasonable software cost estimates. Most of these methods were described by Barry Boehm [Boehm81] in 1981. There has been some improvement in the methods since then, but generally his description of the methods are still valid today.

1. *Algorithmic cost modeling* — A model is developed using historical cost information which relates some software metric (usually its size) to the project cost. An estimate is made of that metric and the model predicts the effort required.
2. *Expert judgment (wideband-delphi)* — One or more experts on the software development techniques to be used and on the application domain are consulted. They each estimate the project cost and the final cost estimate is arrived at by consensus.
3. *Estimation by analogy* — This technique is applicable when other projects in the same application domain have been completed. The cost of

a new project is estimated by analogy with these completed projects. Myers gives a very clear description of this approach

4. *Parkinson's Law* — Parkinson's Law states that work expands to fill the time available. In software costing, this means that the cost is determined by available resources rather than by objective assessment. If the software has to be delivered in 12 months and 5 people are available, the effort required is estimated to be 60 person-months.
5. *Pricing to win* — The software cost is estimated to be whatever the customer has available to spend on the project. The estimated effort depends on the customer's budget and not on the software functionality.
6. *Top-down estimation* — A cost estimate is established by considering the overall functionality of the product and how that functionality is provided by interacting sub-functions. Cost estimates are made on the basis of the logical function rather than the components implementing that function.
7. *Bottom-up estimation* — The cost of each component is estimated. All these costs are added to produce a final cost estimate.

There are several other methods that Boehm did not mention, function point, feature points (similar to function points), and fuzzy logic.

To make a function-point estimate, you review the requirements and the count the numbers of each type of function (input, output, inquiries, data files, interface) the program will likely need. You then enter these numbers in a table and multiply them by the weights (historically determined) to produce the total number of functions points in each category. The function point sum is multiplied by a complexity factor. The complexity factor is the sum of influence factors (for example, data communications, performance, reusability distributed functions) multiplied by an adjustment factor [Dreger89].

The function point procedure is not very intuitive. According to Watts Humphrey:

"As useful as they are, function points are not fully satisfactory for two reasons. First, they cannot be directly measured and second, they are not sensitive to implementation decisions."

However, the function point method is backed by an international special-interest group called, International Function Point Users Group (IFPUG). This group is continually refining the method and providing guidelines and standards to those that are interested.

Feature points were derived from function points by Capers Jones. Although, the procedures are similar, Jones tried to overcome the direct measure problem by providing a number of function-point conversion factors that permit you to count lines-of-code and calculate the program's likely function point content [Humphrey95].

Lawrence Putnam [Putnam92] described the fuzzy-logic estimating method, where estimators assess a planned product and roughly judge how its size compares with historical data on prior products. The problem with this method is that you need a considerable amount of historical data. Another problem is the size of programming applications has historically grown by about an order of magnitude every 10 years [Humphrey95]. If your previous products are increasing in size, it makes product comparisons very difficult.

Each estimating method has its merits and in some cases the project manager should use two or more methods to overcome the deficiencies inherent in each method. Lawrence Putnam [Putnam92] recommends:

> To meet these various [estimating] needs requires multiple approaches.
>
> At times only one approach is applicable. At other times several approaches may be relevant. In the latter case the result of each approach is combined by a weighted statistical process, resulting in a bounded size estimate.*

* Two primary techniques are employed: Bayesian weighting and exponential smoothing.

> The purpose of these quantitative methods is to bound the size, determine the degree of uncertainty of the estimate, and identify the amount of risk associated with the estimate. The multiple approaches enable the organization to view the sizing problem from different perspectives. The statistical techniques provide a final estimate that is more narrowly bounded and represents a lower degree of risk than any single method would permit. Continued use of the methods enables the organization to refine the estimate, or bounds, as further information becomes available.
>
> If significant changes take place from a previous estimate to the current one, however, the exponential smoothing technique may not be sensitive enough to compensate for the amount of the change. In that case, the new estimate should be treated as a new starting point.

As the software project estimates become clearer, the project manager can then start the planning processes. The software project plan is the result of these processes. The project plan identifies the project tasks (processes), resources, responsibilities, and risks, and delivery products.

It is beyond the scope of this paper to discuss each attribute contained within the software project plan. I am going to briefly discuss configuration management, quality control / management, documentation, and project team issues.

Another major problem facing software development is version control and development environment stability. These tasks are accomplished by the configuration

---

Bayesian weighting is an averaging technique that gives more weight to those expected values of the size that have the least amounts of uncertainty. In other words, wild guesses are given less weight in arriving at the final estimate than reasonable, or narrowly bounded, estimates. The degree of uncertainty associated with any estimate is quantified by its standard deviation.

This weighting technique is used both within each estimating method and to combine the results of the different methods. The result is that at each point in the estimating process the uncertainty associated with the estimate at that point has been reduced. At the ultimate combined estimate the uncertainty is at the minimum consistent with the input uncertainties. A low level of uncertainty is indicative of a low level of risk.

Exponential smoothing is a convergence technique that picks up growth or reducing trends and updates the estimate to reflect those trends. As the software design changes, this technique enables the changes to pass smoothly through the software equation to the time-effort-resources estimates. It permits schedules, budgets, and staffing levels to be updated during the project.

management team. Ian Sommerville [Sommerville92] describes the responsibilities of the configuration management (CM) team:

> The role of the CM team is to ensure that changes are incorporated in a controlled way.
>
> In a large project, a formal document naming scheme should be established and used as a basis for managing the project documents.
>
> The CM team should be supported by a configuration database which records information about system changes and change requests which are outstanding. Projects should have some formal means of requesting system changes.
>
> System building is the process of assembling system components into an executable program to run on some target computer system.
>
> When setting up a configuration management scheme, a consistent scheme of version identification should be established.
>
> System releases should be phased so that a release which provides new system functionality is followed by a release to repair errors.

The team does a lot more than this. Basically, they control software, documentation, and process definition releases, maintain software baselines (the previous version or versions for software, documentation, standards, process definitions, equipment environment), and distribute builds (completed software modules) to the software system integration team.

Another important group, identified in the software project plan, is the quality assurance team. As the name suggests, this team is responsible for assuring quality within the project and software development organization. They assist the quality control personnel, who are responsible for defect removal, and project managers. Unfortunately, the quality assurance team is not one of the well-respected teams in the software development organization. In most cases the role designated by management is wrong, or the team members are either brand-new software engineers or, worst, assigned because they are poor designers or programmers.

Lowell Arthur [Arthur92] had this observation:

> Known software techniques make defect-free [quality] software possible. Most software professionals, however, avoid doing all of the things required to achieve zero-defect software. "Too much structure," they proclaim. "Too much bureaucracy."

The primary technique utilized by software practitioners is formal and informal technical reviews. The reviews consist of inspections (this is an analysis technique that relies on visual examination of development products) and walkthroughs (this is a technique in which the designer or programmer leads one or more other members of the development team through a segment of design or code that he or she has written). These two review processes can eliminate more defects than testing the code. According to Glenford Myers [Myers79]:

> Inspections and walkthroughs have been found to be far more effective, again because people other than the program's author are involved in the process. These processes also appear to result in lower debugging (error correction) costs, since, when they find an error, the precise nature of the error is usually located.
>
> Experience with these methods has found them to be effective in finding from 30% to 70% of the logic design and coding errors in typical programs. ... Uses of code inspections by IBM have shown error-detection efficiencies as high as 80% (not 80% of all errors, because we can never know the total number of errors in a program, but in this case 80% of all errors found by the end of the testing processes).

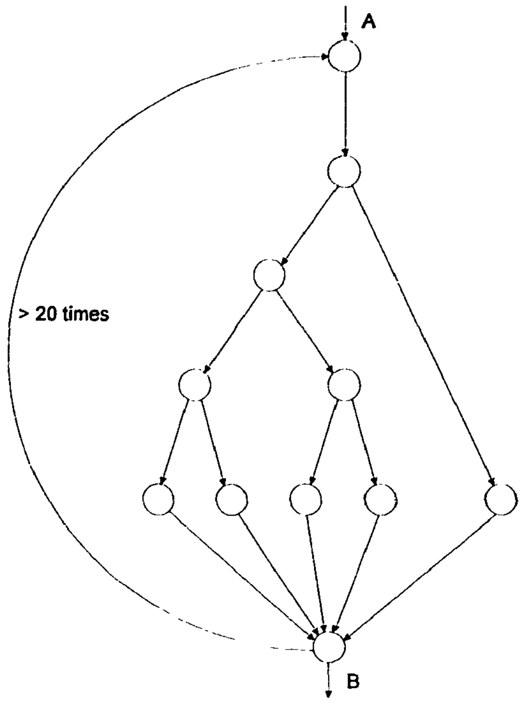
Lowell Arthur [Arthur92] observed:

> Most software companies foolishly base all of their defect efforts on finding bugs once they're in the software. Computer testing to identify and remove defects will find at most 70 percent of the defects. The number of defects that slip through testing is a function of the number of defects in the software

> when it is delivered for testing. The number of defects in the software when delivered to testing is a direct function of the quality of the process used to create the software. Testing can only uncover 70 percent of the latent defects in the code. Inspections can remove 80 to 90 percent of the defects before testing, but a good process will prevent defects from ever entering the product.

Arthur's observation is extremely important. We need to eliminate the majority of software defects (requirements, design, code structure, code logic, etc.) before it ever arrives for testing. It is impossible to find all the defects in the testing process. If we eliminate the defects prior to testing (which are usually the defects that the test team can not find) we will have a better product. I know of several software development organizations, including the one I am now working at, that do not have a formal review process in place, and in some cases do not feel it is important. These organizations "know" that testing will find all the critical and severe defects that the coders and designer placed in the code. What these organizations don't realize is the time it takes to uncover defects in testing. Glenford Myers [Myers92] discusses the time required to test every logic path in a small module.

> The number of unique logic paths through a program is astronomically large. To see this, consider the trivial program represented in the following figure.



The diagram is a control-flow graph. Each node or circle represents a segment of statements that execute sequentially, possibly terminating with a branching statement. Each edge or arc represents a transfer of control (branch) between segments. The diagram, then, depicts perhaps a 10- to 20-statement program consisting of a DO loop that iterates up to 20 times. Within the body of the DO loop is a set of nested IF statements. Determining the number of unique logic paths is the same as determining the total number of unique ways of moving from point **A** to point **B** (assuming that all decisions in the program are independent from one another). This number is approximately 10^{14}, or 100 trillion. It is computed from $5^{20} + 5^{19} + \ldots + 5^{1}$, where 5 is the number of paths through the loop body. Since most people have a difficult time visualizing such a number, consider it this way: if one could write, execute, and verify a test case every five minutes, it would take approximately one billion years to try every path.

Remember this is a 10- to 20-statement program. The number of paths for a one million statement program would be impossible to calculate. Let's just say that a testing team cannot find every defect in a program. Inspections and walkthroughs help the testing team by eliminating the defects that are usually the most difficult to find in testing. These defects are the ones that usually kill the unsuspecting user.

Another quality problem is measurement. We do not know how to measure software quality. According to Norman Fenton [Fenton91],

> It would be difficult to imagine how the disciplines of electrical, mechanical and civil engineering could have evolved without a central role for measurement. But it has been almost totally ignored within mainstream software engineering. More often than not:
>
> 1. We still fail to set measurable targets when developing software products. For example we promise they will be "user-friendly", "reliable", and "maintainable" without specifying what these mean in measurable terms. This prompted the assertion of Gilb:
>
>   **Gilb's principle of fuzzy targets:** *Projects without clear goals will not achieve their goals clearly.*
>
> 2. We fail to measure the various components which make up the real costs of software projects. For example, we usually do not know how much time was really spent on design compared with testing.
> 3. We do not attempt to quantify the quality (in any sense) of the products we produce. Thus, for example, we cannot tell a potential user how reliable a product will be in terms of likelihood of failure in a given period of use, or how much work will be needed to port the product to a different machine environment.
> 4. We still rely on purely anecdotal evidence to convince us to try yet another revolutionary new development technology or tool.

Fenton continued with this discourse and provided some valuable insight on the types of measures that managers and software engineers can perform during the development phases:

Managers:

- Need to measure the *cost* of various processes within software production. For example, the process of developing a whole software system from the listing requirements stage to maintenance after delivery has a cost which must be known in order to determine its *price* for suitable profit margins.
- Need to measure the *productivity of* staff in order to determine pay settlements for different divisions
- Need to measure the *quality* of the software products which are developed, in order to compare different projects, make predictions about future ones, and establish baselines and set reasonable targets for improvements.
- Need to define measurable targets for projects like how much test coverage should be achieved and how reliable the final system should be.
- Need to measure repeatedly particular process and resource attributes in order to determine which factors affect cost and productivity.
- Need to evaluate the efficacy of various software engineering methods and tools, in order to determine whether it would be useful to introduce them to the company.

Engineers:

- Need to monitor the quality of evolving systems by making process measurements. These might include the changes made during design, or errors found during different reviewing or testing phases.
- Need to specify quality and performance requirements in strictly measurable terms, in order that such requirements are testable. For example a requirement that a system be "reliable" might be replaced

by "the mean time to failure must be greater than 15 elapsed hours of CPU time".

- Need to measure product and process attributes for the purpose of *certification.* For example, certification may require measurable properties of the product e.g. "less than 20 reported errors per β-test site", "no module more than 100 lines long", or of the development processes e.g. "unit testing must achieve 90% statement coverage".
- Need to measure attributes of existing products and current processes to make predictions about future ones. For example i) measures of "size" of specifications can be used to predict "size" of the target system, ii) predictions about future maintenance "blackspots" can be made by measuring structural properties of the design documents, and iii) predictions about the reliability of software in operational use can be made by measuring reliability during testing.

The software managers and software practitioners need to document their measurement techniques and results. This is another problem with the current software development methods, poor documentation. Ambiguous documentation or no documentation at all will decrease the likelihood of ever managing the software processes. If you don't know what you have done during the process, how can you recognize the possible eventualities when you start a new project. Lem Ejiogu [Ejiogu91] observed:

> The goal of documentation is communication both during and after the project. During the project, documentation aims at eradicating misunderstanding or distortion of ideas by recording exactly what is visibly accomplished or perceived but yet deferred. After completing the project, documentation "records the history of development, serves as a tutorial guide to system operation, demonstrates that the program works, and provides a means for maintenance and evaluation of obsolete or amendable portions of

> the system.... To be effective, documentation has to have purpose, content, and clarity."
>
> This, in a nutshell, summarizes the problems of documentation in software engineering. If there were nothing like maintenance, documentation would have been, perhaps, less prominent than it is now. But maintenance is an essential and on-going activity of any engineering discipline. ... Some installations have the good habit of not keeping or updating (yes, doing maintenance on) their documentation. This "sin" is aggravated by the turnover of maintenance professionals.

Documentation is a discipline that is extremely important for the production of high quality software. This is an activity that the project manager must encourage throughout the software development effort. Although, most software managers and practitioners look upon documentation as the less glamorous activity.

Documentation would not be a problem, if the software managers and practitioners understood its importance to the software organization. Of course, some do know this and don't care because they will not be working with the maintenance team. They have the attitude of let's get the thing working so we can go onto the next project. A good manager can overcome this difficulty if he fully understands group dynamics and team politics. If the software practitioners (engineers), working as part of the team, better understood group dynamics they can achieve more than they can working alone. According to Ian Sommerville [Sommerville92]:

> An understanding of group dynamics helps software managers and engineers working in a group. Managers are faced with the difficult task of forming groups. They must ensure that the group has the right balance in both technical skills and experience and in terms of personalities.

Sommerville also added"

> Software engineers working in groups can achieve better results and more harmonious working conditions if they understand how the group members

interact and how the group, as a separate entity, takes its place within an organization.

Sometimes individuals working in a group work well together and sometimes they clash so dramatically that little or no productive work is possible.

Sommerville went on to explain the individual orientation types:

Very roughly, individuals in a work situation can be classified into three types:

1. **Task-oriented** — This type is motivated by the work itself. In software engineering, they are technicians who are motivated by the intellectual challenge of software development.
2. **Self-oriented** — This type is principally motivated by personal success. They are interested in software development as a means of achieving their own goals. Attaining these goals may mean that they will move away from technical software development into management.
3. **Interaction-oriented** — This type of individual is motivated by the presence and actions of co-workers. Until recently, there probably weren't many individuals of this type involved in software development because of the apparent lack of human interaction involved in the process. However, as software engineering becomes more user-centered, interaction-oriented individuals may be attracted to software development work.

Besides these orientation types there are four other "attitude" types; the dabbler, hacker, compulsive, and master. Lowell Arthur [Arthur93] defined the four types plus explained how masters become masters:

Dabblers go from one thing to another, never resting long enough to become proficient at any one thing.

Hackers develop a low level of proficiency and then are content to stay mediocre the rest of their lives. All software "hackers" fall into this category. The programmer whose programs are always in trouble; the software "genius" who runs around fighting fires; and the manager who puts up with this type of behavior are all hackers.

Compulsives are not content to be on the plateau of learning. They cram course after course into their lives. Ultimately, they just burn out.

Masters, on the other hand, know that the path to mastery is a journey. To become a master, they must always have a "beginner's mind" that is open and receptive to new learnings. They rise to the first level of skill and then they are ready to practice until they experience the next burst of learning and rise to a new level. Mastery is the path of kaizen — continuous incremental improvement in our skills and abilities. Masters recognize and understand kaizen and Shewhart's Plan, Do, Check, and Act (PDCA) method because they have done it all of their lives to achieve mastery.

There are five keys to software mastery — instruction, practice, surrender, intent, and pushing the envelope.

- Get instruction. Quality begins with training and ends with training.
- Practice. Learning the quality tools requires practice.
- Suspend your disbelief about what will or won't work, and surrender to your practice.
- Develop a clear intent to be the best. Intent is not hope. Hope offers only the flimsy wish to become excellent. Intent is a clear, definite desire and direction.
- Take a risk, push the outside of the envelope. Once firmly grounded in the basics, masters push the limits of what they know to enable them to learn more about what works and what doesn't.

The software manager and software engineer master is one who is disciplined, who have vision, known capability, and in most cases the potential capability of their peers. The masters practice individual process management. They strive to be the best and at the same time share what they have learned with others so that they too can become masters. Watt Humphrey [Humphrey95]:

> We each have responsibilities to others and to ourselves. We need to understand our own abilities, to apply them to our assigned tasks, to manage our weaknesses, and to build on our strengths. While we should do this as part of our everyday work, it is also our responsibility to ourselves. We are each blessed with unique talents and opportunities. We need to decide what to do with them.
>
> Consistent high performance takes persistent effort, an understanding of your own abilities, and a dedication to personal excellence. World-class runners know their best time for a mile, and they know the world record. They know it would make no sense to strive for a 3:00 mile but that 3:40 may soon be achievable. Decades ago, the 4:00 mile was thought beyond human capability. Roger Bannister proved that wrong in 1954. While beating a world record is more challenging than ever before, people keep doing it. They don't do it blindly, however. They develop aggressive personal goals and work ceaselessly to achieve them. When they achieve them, they then pick more aggressive goals and start all over again.

The software industry is seeking out these masters, jot for their wisdom, dedication, insight, and capability to motivate and train others to be masters. No, they seek them out, to help put the organization's fires out. The software industry is still under the illusion that if you find the best software engineers, they will produce the best work. Unfortunately it doesn't work that way. According to Watts Humphrey [Humphrey95]:

> There is a common view that a few first-class artists can do far better work than the typical software team. The implication is that they will know intuitively how to do first-class work, so no orderly process framework will be

> needed. If this were true, one would expect that those organizations who have the best people would not suffer from the common problems of software quality and productivity. Experience, however, shows that this is not the case. A few of the nation's leading software organizations have consistently hired the top graduates from the best computer science schools. They are thus staffed with the best available people, yet their programming groups have many of the same problems that plague everyone else. It seems that the super programmer approach requires better people than are available, even from our leading universities. While this may be a theoretical solution, it is clearly not a practical one. Attracting the best people is vital, but it is also essential to support them with an effectively managed software process.

Software organizations that use "first-class" software engineers without an effective managed software process are operating in a pure chaotic environment. These organizations perceive that they have control of their software production, but it is only a smoke screen. Watts Humphrey earlier in Humphrey89 stated that software organizations will not survive in the 90s or the earlier part of the next century without a managed software process. If these organizations do not evolve their software development processes into mature managed processes, they will not able to compete with other more mature organizations. They will disappear as did the automobile manufactures in the late 1960s and 1970s.

Today's software organizations need to be aware of process engineering and process management which includes risk assessment and risk management. The "traditional" management styles used the last few decades will not work with the rapid advances in technology and communications. Software organization need to understand the whys and hows of process management and how to move from current process models (which are not working) to mature process models.

Rosalind Ibrahim [Ibrahim95] expanded on this by discussing problems and possible solutions:

Today's software organizations are striving to remain competitive and healthy. One path to providing a competitive edge lies in establishing an organizational culture driven by quality aspirations and continuous improvement. For such organizations it is necessary that software engineers and managers are properly equipped to implement improvements and changes. The challenge for educators and trainers is to ensure that adequate knowledge and skills are acquired so that organizations can make rational decisions and carry them out effectively, i.e., to ensure that the organization possesses a solid base of competency in process improvement.

Software engineering organizations tell us that they encounter obstacles to process improvement such as the following:

- "lack of awareness and understanding"
- "inadequate training"
- "misunderstanding of the importance of process improvement"

Some of the needs and recommendations we have heard include the following:

- "We must educate people on the process so that they understand why we're doing this as opposed to just getting a good grade."
- "Educate/train people from the top down and from the bottom up."
- "Get process improvement exposed more in commercial/educational organizations."
- "Include process improvement in formal software education curriculum."

We hope to help overcome these obstacles and start meeting these needs by examining what process improvement education and training entails.

Process improvement is an emerging topic in software engineering education and training. It is so new that the body of knowledge is still evolving, yet there are considerable data available regarding what one might need to know. They can be found scattered in various courses, tutorials, workshops, documents, articles, curricula, standards, texts, etc. They are known by those who are working on process improvement in the field, but they have not been compiled to help software engineering educators and trainers offer the requisite knowledge and skills their students need.

Another software engineering expert, Neal Whitten [Whitten95] also described today's software development organization predicament:

> Many software development organizations do not fully embrace a defined, repeatable, and predictable software development process. Without a disciplined process, they usually face a significantly increased risk in predicting and controlling the critical factors of schedule, cost, function, and quality. So why, then, do many organizations operate without an acceptable software development process?
>
> In some cases the organization may have currently defined processes, but those processes are ineffective for one or more of the following reasons:
>
> - *Not comprehensive enough*: They do not already define all of the activities that apply to all new projects.
> - *Overly complex*: They require too much time and skill to comprehend and apply.
> - *Not flexible*: They are not easily tailored to meet the unique needs of new projects.
> - *Not "owned"*: There is weak or no buy-in from the project's members.
> - *Not understood*: The project's members have not been trained sufficiently.
> - *Not continuously improved*: Lessons learned from past projects are not used to improve the current processes.

- *Not enforced*: The guidelines are there, but the project leadership lacks the discipline to enforce them.

"Process improvement" and "continuous improvement" are terms that are used to describe organizations that are in the process of maturing from ad-hoc operations to repeatable and defined operations. Rosalind Ibrahim [Ibrahim95] continued with her discussion by describing the concepts of process maturity:

> Processes can be characterized in terms of capability, performance, and maturity.
>
> **Software process maturity.** The extent to which a specific process is explicitly defined, managed, measured, controlled, and effective. The maturity of an organization's software process helps to predict a project's ability to meet its goals.
>
> **Software process capability**. The range of expected results that can be achieved by following a software process. A more mature process has improved capability (a narrower range of expected results).
>
> **Software process performance**. The actual results achieved by following a software process. A more mature process has improved performance (lower costs, lower development time, higher productivity and quality) and performance is more likely to meet targeted goals.
>
> **Maturity model**. A representation of the key attributes of selected organizational entities which relate to the progress of the entities towards reaching their full growth or development.
>
> **Institutionalization.** Building an infrastructure and a corporate culture that supports the methods, practices, and procedures of the business so that they endure after those who originally defined them have gone; an organization institutionalizes its software process via policies, standards, and organizational structures.

Edward Deming expanded upon Shewhart's work and started the process improvement and maturity practices in Japan during the 1950s. He called those processes that were understood and documented as "defined processes." Deming [Deming86] defined these processes as "something everyone can communicate about and work toward." Define processes provide the following benefits:

- They enable effective communication about the process among users, developers, managers, customers, and researchers.
- They enhance management's understanding, provide a precise basis for process automation, and facilitate personnel mobility.
- They facilitate process reuse. Process development is time consuming and expensive. Few project teams can afford the time or resources to fully define the way they will work. They can save both by using the standard reusable elements a defined process provides.
- They support process evolution by providing an effective means for process learning and a solid foundation for process improvement.
- They aid process management. Effective management requires clear plans and a precise, quantified way to measure status against them. Defined processes provide such a framework.

These enablers cannot be put in place in a traditional management system. Deming's management principles are quality management. His management principles were primarily developed for manufacturing organizations. He also incorporated Shewhart's work of the 1930s into his management system.

Shewhart's classical management strategy provides a orderly approach to controlling and improving quality by studying a process and analyzing its performance through four steps: *plan*, *do*, *check*, and *act*. This strategy can be applied at various process levels, and several improvement approaches are derived from this basic cycle.

Plan. Define the problem; state improvement objectives.

Do. Identity possible causes of the problem; establish baselines; test change.

Check. Evaluate; collect data.

Act. Determine effectiveness; implement system change.

Deming's and Shewhart's work is incorporated in a process model that is just starting to take the software industry by storm. The researchers for this model have researched hundreds of software development organizations, They looked for the common denominator for all the successfully completed projects. What they found were 18 key process areas with over 250 activities. This research was sponsored by the Department of Defense and performed by the Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania. The model is called the Software - Capability Maturity Model or CMM. Rosalind Ibrahim [Ibrahim95] describes:

> The CMM applies process management and quality improvement concepts to software development and maintenance. It is a model for organizational improvement and serves as a guide for evolving toward a culture of engineering excellence. The CMM provides the underlying structure for software appraisals assessments and evaluations. It offers a staged improvement structure based on the quality principles of Deming, Juran, and Crosby.
>
> **Critical concepts**. Software process: process capability, process performance, process maturity, and institutionalization.
>
> **Structure and components of the CMM.** Maturity levels indicate process capability and contain key process areas. Key process areas achieve goals and are organized by common features. Common features address implementation or institutionalization and contain key practices. Key practices describe infrastructure or activities that contribute to satisfying the goals of that key process area.

The maturity levels. Each level is a well-defined evolutionary plateau toward achieving a mature software process; each level builds a foundation for succeeding levels to use to implement process effectively and efficiently.

Level 1: Initial. Process is informal and ad hoc; performance is unpredictable.

Level 2: Repeatable. Project management system is in place; performance is repeatable; and there is a disciplined process.

Level 3: Defined. Software engineering and management processes are defined and integrated; there is a standard, consistent process.

Level 4: Managed. Product and process are quantitatively controlled; there is a predictable process.

Level 5: Optimizing. Process improvement is institutionalized; there is a continuous improvement process.

The CMM has the framework to provide the foundation to establish a well researched and proven process improvement methodology into a software development organization. In fact, the model itself is dynamic and improving with time. The Software Engineering Institute was established to provide software engineering services to Department of Defense organizations, but it is now providing this same service to commercial software development organizations.

This paper provided extensive coverage on what is wrong with our software organizations. Before we go on to how and why software organizations must change the way they do business, I would like to add this description of a "real" project manager by Tom DeMarco [DeMarco87].

In my early years as a developer, I was privileged to work on a project managed by Sharon Weinberg now president of the Codd and Date Consulting Group. She was a walking example of much of what I now think of as enlightened management. One snowy day, I dragged myself out

> *of a sickbed to pull together our shaky system for a user demo. Sharon came in and found me propped up at the console She disappeared and came back a few minutes later with a container of soup. After she'd poured it into me and buoyed up my spirits, I asked her how she found time for such things with all the management work she had to do. She gave me her patented grin and said, "Tom, this is management."*
>
> Sharon knew what all good instinctive managers know: The manager's function is not to make people work, but to make it possible for people to work.

Organizational Evolution

Chaos, is the tar pit Fred Brooks [Brooks82] described in "The Mythical Man-Month." The tar pit turned out to be an apt analogy because so many extinct species of animals can now be found in the La Brea tar pits, and those software development organizations that remain in chaos will ultimately drag themselves and their company down into the software tar pit. Chaos causes cost overruns and project failures and customer anger and alienation. At this point, managers and programmers don't trust each other; fire fighting reigns supreme, and there is no time to think about how to do things better. It seems that the software development organizations keep slipping back into the tar pit. Software creation and evolution demands that these organizations begin pulling themselves up out of the tar pit and slowly, one stair at a time, begin moving up the stairway to order and software excellence, a world-class organization.

Software development organizations need to look within the organization to find the resources and skills necessary to become a world-class organization. Lowell Arthur [Arthur93] described the primary reason for change:

> To maximize productivity and quality, we will need to reduce the causes of poor quality: procedures and methods, materials, environment, people, and external factors.

Arthur continued with a description of the key components in any software development organization, people, process, and technology:

> In our flutter from one silver bullet to the next, in our search for a savior or a magic wand, we have overlooked the obvious or pooh-poohed them as too simple. We must become more like the tortoise and less like the hare; we must seek continuous progress toward our goal. Then, in our journey, if we run across a silver bullet or a magic wand, we'll know what to do with it.
>
> To climb to the top of the stairway to software excellence, we must implement the following key elements of people, process, and technology:
>
> **People** We must establish several key specialist groups:
>
> - Quality improvement specialists to get quality rolling
> - A process team to improve our processes
> - A reengineering team to continuously improve our software and data
> - A measurement team to see how we're doing
> - An estimating team to improve our estimates
>
> **Process** We need methods that must be continuously improved to reduce our time-to-market for applications, products, and services:
>
> - A flexible development methodology
> - A defined evolution methodology
>
> **Technology** We need:
>
> - A full-blown maintenance workbench including re-engineering tools
> - A fully integrated development workbench that supports the methodology

- A change management system
- A suite of measurement tools

Capers Jones [Jones91] provided this observation as a consultant to many Fortune 500 organizations:

> These five steps to software quality control have been observed in the course of software management consulting in leading corporations:
>
> **Step 1 Establish a software quality metrics program**
>
> Software achieved a notorious reputation during the first 45 years of its history as the high-technology occupation with the worst track record in terms of measurements. Over the last 10 years, improvements in measurement technology have enabled leading-edge companies to measure both software quality and productivity with high precision. Quality measurement is a critical factor in high-technology products, and all the companies which have tended to become household words have quality measurement programs: DEC, Hewlett-Packard, IBM, and many others. The lagging enterprises which have no software measures also have virtually no ability to apply executive control to the software process.
>
> **Step 2 Establish tangible executive software performance goals**
>
> Does your enterprise have any meaningful software quality or productivity goals operational? The answer for many U.S. companies would be no, but for leading-edge companies such as IBM and Hewlett-Packard it will be yes. Now that software can be measured, it is possible to establish tangible, pragmatic performance goals for both software quality and productivity. Since the two key aspects of software quality are defect removal efficiency and customer satisfaction, reasonable executive targets would be to achieve higher than 95 percent efficiency in finding software bugs and higher than 90 percent “good” or “excellent” customer satisfaction ratings.

Step 3 Establish meaningful software quality assurance

One of the most significant differences between leading and lagging U.S. enterprises is the attention paid to software quality. It can be strongly asserted that the U.S. companies that concentrate on software quality have higher productivity, shorter development schedules, and higher levels of customer satisfaction than companies that ignore quality. Since the steps needed to achieve high quality include, both defect prevention and defect removal, a permanent quality assurance organization can facilitate the move toward quality control.

Step 4 Develop a leading-edge corporate culture

Business activities have a cultural component as well as a technological component. The companies that tend to excel in both market leadership and software engineering technologies are those whose corporate cultures reflect the ideals of excellence and fair play.

As Tom Peters has pointed out in his landmark book, *In Search of Excellence,* the truly excellent enterprises are excellent from top to bottom. If the top is not interested in industry leadership or doesn't know how to achieve it, the entire enterprise will pay the penalty.

Step 5 Determine your software strengths and weaknesses

More than 200 different factors can affect software productivity and quality; they include the available tools and workstations, the physical environment, staff training and education, and even your compensation plans.

This step is logically equivalent to a complete medical examination in a major medical institution. No physician would ever prescribe therapies without a thorough examination and diagnosis of the patient. The same situation should hold true for software: Do not jump into therapy acquisition without knowing what is right and wrong in all aspects of your software practice.

The last step is very important. Before you can improve, you must know your current capabilities. For a software development organization, this means performing a process assessment to determine the capabilities and also to establish a baseline. According to Watts Humphrey [Humphrey89]:

> Assessments are done:
>
> - To learn how the organization actually works
> - To identify its major problems
> - To enroll its opinion leaders in the change process
>
> The assessment team leader should be someone with considerable software experience, the ability to lead small groups and the ability to convincingly present the results. The assessment team members should all be experienced software developers.
>
> Senior management must assign sufficient priority to the assessment and improvement effort, or adequate resources will not be assigned and no significant actions will likely result.

Senior management sponsorship is extremely important. However, it is very difficult to overcome comfort zones and old habits. Some senior managers recognize that change is important for others, but not for themselves. Dr. Dennis Jaffe and Dr. Cynthia Scott (*Building a Committed Workplace: An Empowered Organization as a Competitive Advantage*) [Ray93] discussed the underminers that occur during organizational transformation:

> Many organizations see change as something that can be declared, and implemented without much difficulty. They are still operating on a 19th century view of human nature, where people are motivated by appropriate reward and punishment. With money, or the threat of termination, people will

go along. Managers assume that if they order people to change, they will. They do not recognize the tremendous internal struggle, the emotional dynamics, the upheaval, and the nature of the learning process that organizational renewal poses for individual employees. Viewing change from the executive perspective, a study of top executives found that 80% felt their companies had to change. However, only 20% felt that *they* had to change. People see the need for change, but not the immensity of the personal and professional disruption it entails.

Ironically, many of the actions of top managers actually increase alienation, anger, frustration and add to the confusion. They say they want empowerment, but intentionally or most often unintentionally, they produce the opposite. Some of the common underminers of large organizational transformation include:

1. **Incongruence between** the stated goals and what they do (e.g., act directive and controlling, while asking for empowerment).
2. **Emotional Illiteracy,** not understanding the complex emotional dynamics of people faced with drastic and total shifts in the nature of their work.
3. **I Don't Have to Change, "They" Do.** Feeling that the leader is an exception, and not available and open to learning.
4. **Not Giving Up Control.** Empowerment is accompanied by trust, rooted in understanding that the leader alone can't solve the problems. Employees need to be allowed to come up with innovations, and trust in their goodwill.
5. **Isolation.** The leader doesn't come out of his office and doesn't really seek out and listen to distressing information from employees. One of the easiest ways leaders maintain illusions is by staying on the phone with the central office, traveling a lot, and relying on subordinates to tell them

what's going on. The essence of this behavior is fear of listening, and inability to manage people in distress.

6. **No Models of New Behavior** — People can't be ordered to change, give up control, or take more responsibility, if they have never learned how to do it. People need to see and practice new models of behavior, and they need time to learn, and space to experiment and even make mistakes.
7. **Impatience** — Many promising programs are discontinued just as they are on the verge of payoff because the management feels it isn't working, or worse, they find a new fad or program and move on to that. The key factor in successful change seems to be persistence in a direction, with prudent feedback and course correction along the way.
8. **Middle Management Entrenchment** — Middle managers are an endangered species, and in many change efforts they are the most threatened They are expected from above to produce results, and they feel the pressure of newly empowered, newly competent people from below Under threat, they dig in. They need support, security, and help in learning new ways.
9. **Failure to Understand People's Needs for Psychological Security** — Change is terrifying and the company needs to provide some form of psychological security. That does not mean job security, which doesn't exist, but at least offer clear information on what is happening, options and possibilities, and then allowing people the time to move through the phases of transition.

Those who fall into these traps have elaborate theories of who's to blame: unmotivated employees, lack of resources, corporate policies, bad competitors, or the economic pressures. But the truth, which they deny and avoid, is that the major obstacle to change in many large companies is the lack of self-awareness in top management, their lack of capacity to see that they

> themselves need to change in ways they at first do not fully comprehend, that they need to let go of control and allow the power of the people below them to grow.
>
> Empowerment requires a deep transformation of management style. Managers need to shift their attention from other people to themselves. It takes time, conscious and careful planning, and a series of steps that teach people new ways and move toward new structures.

Dr. Jaffe and Scott discuss the benefits of empowerment in an organization. However, most of the organizations that I have consulted with do not fully understand the "empowerment" concept. Some organizations confuse the term with delegation and permissiveness. In some cases responsibility is delegated without authority, and this is called empowerment. In other cases individuals are "empowered", yet are penalized for taking appropriate risks. This subject is discussed more by Jim McCarthy [McCarthy95]:

> Although I long for another word to describe [empowerment] because "empowerment" has become so debased in contemporary usage, empowerment by any name has to be a central value in any group creating intellectual property. We often confuse permissiveness with empowerment. But enabling people to do whatever they think best is very different from enabling them to think and do their best.
>
> And to empower someone is to enable them to be their best, is to free them from the infinitely varied kinds of blockages that tend to plant themselves in the path of accomplishment in the untended organization. Freedom is the cornerstone of empowerment, freedom to develop and apply judgment, freedom to think and say what needs thinking and saying, freedom to take risks without extraneously punitive consequences.
>
> Empowerment is the result of teaching and learning, not of neglect and anarchy. For a manager to say to a subordinate, "This is your decision" is empowering only when the manager has provided and continues to supply

what's needed to make a good decision — training, information, adequate resources of whatever stripe. Otherwise, such a delegation is really a dereliction.

If everybody is empowered, how are decisions made when there's conflict? This is really more a theoretical than a practical problem. In a properly empowered environment, the situation is not anarchic and confrontational but is meritocratic. As people become secure, they abandon much of the foolishness that stems from weak egos. Devoid of ego pathology, most design, development, and organizational decisions are pure resource tradeoffs. An empowered team is capable of analyzing the pluses and minuses of all potential approaches and of optimizing in the interests of a particular shared goal or vision. There is no right approach or wrong approach. There is a continuum of trade-off among features, resources, and time.

Along the same train of thought as Jim McCarthy, Tom Gilb [Gilb88] established a Bill of Rights that formally establishes the rights of empowered individuals:

1. You have a right to know precisely what is expected of you.
2. You have a right to clarify things with colleagues, anywhere in the organization.
3. You have a right to initiate clearer definitions of objectives and strategies.
4. You have a right to get objectives presented in measurable, quantified formats.
5. You have a right to change your objectives and strategies, for better performance.
6. You have the right to try out new ideas for improving communication.
7. You have the right to fail when trying, but also to kill failures quickly.
8. You have a right to challenge constructively higher-level objectives and strategies.

9. You have a right to be judged objectively on your performance against measurable objectives.

10. You have a right to offer constructive help to colleagues to improve communication.

As important as it is for software development organizations to change with the times, the senior managers and software managers and practitioners will not participate with the change if they do not see value for themselves. As Stephen Covey [Covey94] said:

> Make it comfortable to leave the comfort zone and uncomfortable to stay in it.

The other important factor in change is that the individuals must feel that they are part of the change; that they contributed value to the process. This requires managers with insight and vision. As a great Chinese philosopher, Lao-Tzu stated:

> The bad leader is he who people despise. The good leader is he who the people praise. The great leader is he who the people say, "We did it ourselves."

The great leader will accept and promote the changes required and motivate and excite their staff about the changes. There are recommended steps to bring about the organization's evolution. These change requirements are described by Watts Humphrey [Humphrey89]:

> [There are] six requirements for software process change:
>
> 1. *Sell top management*
>   Significant change requires new priorities, additional resources, and consistent support. Senior managers will not provide such backing until they are convinced that the improvement program makes sense.
>
> 2. *Get technical support*
>   This is best obtained through the technical opinion leaders. Every organization has a few technical professionals whose opinions are widely

respected. When they perceive that a proposal addresses their key concerns, they will generally convince the others. On the other hand, when the technical community is directed to implement something they don't believe in, it is much more likely to fail.

3. *Involve all management levels*

   While the senior managers provide the resources and the technical professionals do the work, the middle managers make the daily decisions on what is done. When they don't support the plan, their priorities will not be adjusted, and progress will be painfully slow or nonexistent.

4. *Establish an aggressive strategy and a conservative plan*

   While senior management will be attracted by an aggressive strategy, the middle managers will insist on a plan that they know how to implement. It is thus essential to be both aggressive and realistic. The strategy must be visible, but the plan must provide frequent achievable steps toward the strategic goals.

5. *Stay aware of the current situation*

   It is essential to stay in touch with current problems . Issues change, and elegant solutions to last year' s problems may no longer be pertinent. While important changes take time, the plan must keep pace with current needs.

6. *Keep progress visible*

   People easily become discouraged if they don't see frequent evidence of progress. Advertise success, periodically reward the key contributors, and maintain enthusiasm and excitement.

As the software development organizations make these changes they will begin to recognize their potential to produce reliable and high quality products faster without excessive effort by the software development staff.

"To Go Where No One Has Gone Before"

The key to future software development efforts is measurement. According to David Card [Card90]:

> Measurement will become more important in the future of software engineering. Current software research focuses on developing alternative life cycle paradigms and on design automation through expert systems and artificial intelligence. However, the success of both endeavors depends to some extent on improved measures.
>
> Reliable measures also are essential to design automation. Major software research centers are investing their long-term hopes for software process improvement in artificial intelligence and expert systems. However, no process can be automated before it is well understood, measurable, and controllable. We have to achieve "natural" intelligence before "artificial" intelligence becomes meaningful. Design automation cannot occur without effective design measurement techniques.

We have to know ourselves before we can grow and expand our abilities. Solid measurement techniques will allow the software development organization to know itself and expand its capabilities.

Peter Senge [Senge90] discussed a direction that few software engineering researchers are pursuing, software development simulations. Peter Senge calls this simulation "microworld." Although, Senge's view of the microworld may not be entirely suitable for the software industry, it is, however, plausible. After all, twenty years ago no one imagined the capabilities of the flight simulators we are now using. Back then if you told a pilot that he could obtain a commercial pilot license without setting foot in a real aircraft, he or she would have laughed at you. Today this is common place. The aircraft simulators do a better job training flying techniques than the actual aircraft. Senge's microworlds could do the same with the use of expert systems and "artificial" intelligence. This is Peter Senge's [Senge90] view of the microworld:

> Microworlds enable managers and management teams to begin "learning through doing" about their most important systemic issues. In particular, microworlds "compress time and space" so that it becomes possible to experiment and to learn when the consequences of our decisions are in the future and in distant parts of the organization.

Senge continued by describing the issues that are now being studied by his research team and others:

> **Integrating the microworld and the "real" world**
>
> The unique power of microworlds lies in surfacing hidden assumptions, especially those lying behind key policies and strategies, discovering their inconsistency and incompleteness, and developing new, more systemic hypotheses for improving the real system. How can such learning lead to more carefully designed "real life" experiments to test insights gained in microworlds, and will these experiments, in turn, allow managers to design better microworlds?
>
> **Speeding up and slowing down time**
>
> In microworlds, the pace of action can be slowed down or speeded up. Phenomena that stretch out over many years can be compressed to see more clearly the long-term consequences of decisions. We often also want to slow down the interactions among members of the team, so that they can see subtle ways in which they shut down inquiry or discourage testing of different views. Will repeated experiences in microworlds expand managers' perceptual "time window," making them both more perceptive of slow, gradual organizational and business changes and of very rapid interpersonal interactions and thought processes?
>
> **Compressing Space**
>
> In microworlds, managers can learn about consequences of actions that occur in distant parts of the system from where actions are taken. Will this

help them recognize such consequences in real life and make “the systemic choice”?

Isolation of variables

In laboratories, scientists can eliminate intruding outside variables and carefully simplify the complexity of real processes. The real world of management offers no such control; but a microworld is a controlled environment, in which experimenters can ask “What if?” questions about outside factors. Microworlds also let you bring in potential outside factors that have not yet taken place in reality — for example, “Suppose regulators forced us to put a ceiling in rates: what might happen to us?” Will microworlds help managers learn to disentangle complex interactions in real setting?

Experimental orientation

Microworlds let teams experiment with new policies, strategies, and learning skills. Actions that cannot be reversed or taken back in real business can be redone countless times in the microworld. Over time, will microworld learning make management teams more open to consider and test wide ranges of hypotheses, and less likely to get “locked in” to particular ways of looking at problems?

Pauses for reflection

Microworld experiments have revealed just how nonreflective most managers are. Despite the ready access to information and controlled experimentation in the computer environment, managers tend to jump from one strategy to another without ever stating clearly their assumptions and without ever analyzing why strategies produce disappointing results. Will learning to explicate assumptions and reflect on outcomes of experiments in microworlds inculcate habits that carry over to real life decision making?

Theory-based strategy

The business practices of most firms are firmly "anchored" to standard industry practices. By contrast, systems thinking and microworlds offer a potentially new basis for assessing policy and strategy. They lead to "theories" of critical business dynamics which can then clarify the implications of alternative policies and strategies. ... Will continued development of microworlds lead to a new approach to strategy development, that is less vulnerable to accepting implicit mediocre industry standards?

Institutional memory

"Learning builds on past knowledge and experiences — that is, on memory," wrote Ray Stata, CEO of Analog Devices, in 1989 in the *Sloan Management Review*. "Organizational memory must depend on institutional mechanisms," rather than on individuals, Stata says, or else you risk, "losing hard-won lessons and experiences as people migrate from one job to another." Will continued research on microworlds and "generic structure" theories of business dynamics lead to a "library of microworlds"? And will such a library, when tailored to the needs of a particular firm, create a significant new form of organizational memory?

The microworlds of today are rough precursors of what microworlds of the future will be like. All the examples cited above would have been impossible only four or five years ago, before the current generation of personal computers with advanced graphics capabilities. The coming years will see dramatic advances in both the availability and capabilities of microworlds for managers.

Peter Senge's final comment about the future of microworlds is exciting:

In the learning organizations of the future, microworlds will be as common as business meetings are in today's organizations. And, just as business meetings

> reinforce today's focus on coping with present reality, microworlds will reinforce a focus on creating alternative future realities.

Another view of the future is described by Roger Pressman [Pressman92]:

> As hardware and software technologies advance, the very nature of the workplace will change. The following scenario provides one vision of a software engineer's work environment during the first decade of the twenty-first century:
>
>> "Good morning," you say as you enter the office.
>>
>> Your workstation screen brightens, a window appears on the screen, an androgynous face appears, and a very human voice says, "Good morning. You have six voice mail messages, two facsimile transmissions, and a list of daily action items. Five development tasks are listed."
>>
>> The face and the voice belong to your "agent," an interface program that performs a variety of sophisticated clerical duties. It has been customized to anticipate your needs, it recognizes your voice, and it can do many things at once—like answer your phone around the clock, look up information, communicate directly with you, and perform other data processing functions. Communication with the agent can be verbal or written, but most people prefer to speak to their agents.
>>
>> "Show me the action items and development tasks," you say.
>>
>> Immediately the list of action items appears on the display and the agent - begins to read the list aloud, highlighting each item as it is read.
>>
>> "Silence please, and hold the list," you interrupt. "While you're holding, check any of the voice mail or fax transmissions for key words."
>>
>> You have just asked the agent to perform an analysis of each incoming message to determine whether it contains any of a set of key words (these could be people's names, places, phone numbers, or topics that you deem

as especially important). As you scan the list of action items on the screen, you see two appointments, a few telephone calls to be made, and an anniversary present to be purchased.

By the time you have scanned the list of action items the agent's face has reappeared on the screen.

It's early and you're not tuned into the work day as yet. Embarrassed (but why should you be, you're communicating with a machine!), you ask, “What did I ask you to do?”

“You asked me to check any of the voice mail or fax transmissions for key words. Would you like a list?”

“Yes, but only those with reference to changes.” A list of messages appears on your screen. You fix on one item from the list and say “Open”. In less than a second, a video camera built into the workstation has tracked your eye movement at the time you said “Open” and the system has calculated which item you were looking at. You begin to read for a few moments and then stop.

“Please find all modules in the Factory Automation System that have been changed in the last month. Store the name of the modules, the source of the change, and the date and generate an action item for me to review them.”

“What version of the system would you like to use?” asks the agent as a scrolling window appears.

“All,” you reply.

“OK,” says the agent.

While you're going through your mail the agent will have an “apprentice” perform the task you requested. That is, the agent “spawns” a task to perform configuration management functions. Within seconds, the agent

> returns to do your bidding. Simultaneously, the first apprentice is searching the CASE repository looking for module names.
>
> “Can I have a word processor?” you ask the agent. A word processing program, not unlike the best that you see today, appears on the screen. You begin dictating a letter (the keyboard or a handwriting tablet can also be used). The text appears on the screen as you speak each word. While you are dictating, you think of something for the agent to do. Using a pointing device, you click on the agent's window.
>
> “I need a source listing for module find.inventory.item. Insert it at the marker I'll note in the text of the document I was working on. Also, call Emily Harrison in system engineering and tell her that I'll be transmitting the document later today.”
>
> “OK,” responds the agent. Apprentices are spawned to generate the table and make the call while you return to your dictation.
>
> The environment implied by the above “conversation” will change the work patterns of a software engineer. Instead of using a workstation as a tool, hardware and software become an assistant, performing menial tasks, coordinating human-to-human communication, and, in some cases, applying domain-specific knowledge to enhance the engineer's ability.

Roger Pressman’s view of the future is not some ten or fifteen years from now. Most of the technology described by Pressman is in use now. A friend of mine, Rob Rapp, bought a speech recognition software package that integrates with several Microsoft Windows software applications including Microsoft Office. After training the speech recognition application his speech pattern for a week, Rob can now verbally open and close applications, open and save files, write documents in Microsoft Word, spell and grammar check and make corrections to his documents. The software and hardware costed approximately $600.00. Rob said that the time saved in typing and other tasks paid for the software application in less than two months. The future is here and most of us don't even know it.

To create, develop, utilize, and manage the evolving hardware and software technologies will require a disciplined approach. All of the current management and software engineering experts firmly believe this. According to Michael Ray [Ray93]:

> We need to be disciplined, to hold scrupulously to higher values, to operate with creativity, compassion, and community, and to become leaders who see the greater good for each other as the motivating force for what we do. In the present world situation we must ask, as Catherine Ingram did in her book *In the Footsteps of Gandhi,* "How should one lead his life in a world of seemingly intolerable suffering?" Or as Winston Churchill put it, "We make a living by what we get. We make a life by what we give." These are the directions for an individual career and for business in general that are necessary in this time of change.

Software Engineering In Academia

This section discusses Software Engineering as an academic discipline. The first sub-section discusses the Software Engineering academic discipline history. The next section discusses the importance of Software Engineering academic research and Software Engineering academic pursuits, and publications. The final section discusses my computer science and software engineering curriculum analysis for National University.

Evolution of CS/SE Degree Programs

Software engineering is an extension of Computer Science. So it is appropriate to discuss the growth of the Computer Science degree programs as well as the emergence of the Software Engineering degree programs.

It all started in the mid-1950s. In the beginning there were mathematics and mathematicians. These were the first programmers and computer scientists. Why, because it took a mathematician to understand machine code, all 0s and 1s. The initial computer science programs were taught either in the academic department of mathematics or engineering.

The first few Computer Science Ph.D. programs appeared about 1961; these were interdisciplinary programs in existing departments rather than separate degree programs. By 1964 there were about a dozen computer science bachelor's degree programs in US universities. Between 1964 and 1968, the number of bachelor's programs grew to nearly 100; master's programs experienced a similar increase. The number of Ph.D. programs grew from about 10 to about 40

According to Gary Ford [Ford92], the publication of the first computer science curriculum recommendation from the Association for Computer Machinery (ACM) was the catalyst for the rapid growth of undergraduate computer science programs between 1964 and 1968.

S. Pollack [Pollack82] also pointed out:

[ACM] Curriculum '68's influence also had a dichotomizing aspect: Its basically mathematical orientation sharpened its contrast with more pragmatic alternatives. Most computer science educators agreed that the proposed core courses included issues crucial to computer science. However, the curriculum brought to the surface a strong division over the way in which these issues should be viewed. In defining the contents of the courses, Curriculum '68 established clearly its alignment with more traditional mathematical studies, giving primary emphasis to a search for beauty and elegance. Pedagogically, this implied a set of academic objectives concerned chiefly with preparation for graduate study leading to a career in research. Consequently, those colleges and universities holding with the perception of computer science saw Curriculum '68 as a reinforcement and endorsement of their orientation and sought to implement it commensurate with their resources.

On the other hand, many educators felt the curriculum to be at odds with their perception of reality. They argued that the uses of computer science and the observed roles of computer scientists militate for an education approach much closer to that used in professional disciplines. ... In this light, computer science education should have a strong professional flavor with design principles, general approaches to problem solving, and experiments with current methodologies receiving considerable attention. This would be consistent with the expectation of professional employment starting at the baccalaureate level.

S. Pollack [Pollack82] continued by describing the movement toward "research":

The rapid growth of computer science education [in the 1960s] stimulated increased interest in theoretical areas (such as automata theory and formal languages) whose pursuit predated computers. Now, these areas were seen potentially to impinge on questions raised by the design and use of computer systems. Consequently, there appeared to be a prospect of concurrent and mutually nourishing development in computer science theory and practice.

> Curiously, this did not happen. The newly intensified effort generally maintained its own paths, interacting very little with the application-motivated problems that were helping to spur headlong advances in hardware and software technology.

John von Neuman, the father of programming languages, made these prophetic comments regarding the curriculum trend toward pure theory:

> As a mathematical discipline travels far from its empirical source, or still more, if it is a second- and third-generation only indirectly inspired from ideas coming from "reality," it is beset with very grave dangers. It becomes more and more purely aestheticizing, more and more purely *l'art pour l'arts.* This need not be bad, if the field is surrounded by correlated subjects, which still have closer empirical connections, or if the discipline is under the influence of men with an exceptionally well-developed taste.
>
> But there is a grave danger that the subject will develop along the line of least resistance, that the stream, so far from its source, will separate into a multitude of insignificant branches, and that the discipline will become a disorganized mass of details and complexities.... [W]henever this stage is reached, the only remedy seems to me to be the rejuvenating return to the source: the reinjection of more or less directly empirical ideas. I am convinced that this is a necessary condition to conserve the freshness and the vitality of the subject, and that this will remain so in the future.

Unfortunately, von Neumann's comments are somewhat valid for the typical computer science curriculum of the past 10 to 20 years. Very few, if any, foundation courses reached back toward the empirical source of building useful artifacts.

Within the last 10 years the computer science degree programs have specialized into many sub-fields, including software engineering. Most of the degree programs are slowly moving away from the research type curriculum toward the practitioner type curriculum. You might say that the computer science degree programs are starting to produce "software engineering technologist." These graduates have a basic software engineering understanding and can provide the technical know how to put a software design together. This "technologist" has a similar role of the electronic engineering technologist, in that he works side-by-side with an engineer.

A report on software engineering undergraduate education described 11 colleges and universities that have significant course sequences in software engineering. As far as the writers of the report know, there is not a program in the United States that uses "software engineering" in its undergraduate degree program name. The current degree programs are either specialty fields or emphases within another degree program. These programs are evolving very slowly. Gary Ford [Ford92] commented on the growth of the software engineering degree field:

> Compared to computer science, the growth of separate software engineering programs has been much slower. Although there have been calls for such programs as early as 1969, no undergraduate programs actually named software engineering have been created. We do believe, however, that the majority of computer science programs now have at least one course in software engineering.
>
> We [know that] approximately 25 master's or certificate programs in software engineering have been created between 1978 and 1994—a 16-year span.
>
> [The] growth rate is also much slower than that of computer science.
>
> We do not know of doctoral programs in software engineering that are so-named and separate from computer science programs. Many universities report that increasing numbers of students in computer science doctoral programs are writing dissertations on software engineering topics.

In 1987 a report on software engineering curriculum was published by the Software Engineering Institute. The growth rate of software graduate programs were not as great as the computer science growth rate when the ACM published their curriculum recommendations. Gary Ford [Ford92] observed:

> Although the rate of growth of master's programs increased, the SEI curriculum was not the only factor in that growth. The increasing needs of industry for educational opportunities for software engineers was a major reason for the development of new programs. On the other hand, the majority of programs started since 1987 acknowledge having been influenced by the SEI curriculum. This reinforces our belief that a model curriculum is an important catalyst for creation of new programs.

Another reason for the slow growth is that most of the "software engineering" programs are contained within a computer science department or college. Many computer science educators feel that software engineering is not mature enough to warrant a separate program, department, or college. Others feel that software engineering is a specialty field and should not be taught at the undergraduate level. S. Pollack [Pollack82] commented about similar arguments made by educators in the 1960s about computer science:

> At a more fundamental level, many universities, while convinced of computer science's separate identity, felt that an independent program was premature. For them, computer science was a graduate specialty to be preceded by undergraduate concentration in some established area (not necessarily mathematics or science).

The driver behind the computer science "view point" change for many universities was not from within, but from the computer and software industry. According to Donald Christiansen [Christiansen92c]:

> For years now [computer scientists and software engineers] have been grousing about the courses they are required to take as undergraduates [and graduates]. And their employers have been complaining that new graduates are not ready to go to work as *bona fide* engineers.

Fortunately, colleges and universities are listening to their alumni and the entities that recruit their engineering students.

Gary Ford [Ford92] concluded in the Software Engineering Institute's Software Engineering Undergraduate Education report with these statements:

1. We do not expect software engineering programs to emerge as rapidly as computer science programs did. Much of the software industry still relies on relatively undisciplined development processes and is satisfied with the level of programming skills in graduates of existing computer science programs. As the industry matures, there will be a greater demand for software engineers (rather than programmers), but that is a slow process.
2. Although separate software engineering programs in U. S. universities are appearing first at the graduate level, this is not a requirement. It is possible for a university to develop an undergraduate program without first creating a graduate program
3. The publication by the professional societies of a model curriculum for a bachelor of science in software engineering degree would probably accelerate the growth of such programs significantly. The effects of such a model curriculum would include establishing the credibility of such programs, encouraging authors and publishers to create the needed new textbooks, and providing a basis for future accreditation guidelines for such programs.
4. For the immediate future, the most likely evolutionary path will be the creation of a software engineering track within a computer science program. We hope that schools following this path will learn from examples [of other exemplary schools] and develop introductory courses that serve both software engineering and computer science tracks equally well.

Software Engineering Research and Publications

We still don't know all the complexities involved with software development. The same is true with software research. According to Maurice Wilkes [Wilkes95]:

> As the computer industry has become dominated by software, hardware research has receded into the background. In consequence, the model of [traditional] industrial research ... must be applied with caution in the computer industry. Software research makes no demands for laboratory facilities of the traditional kind nor for people with qualifications in the experimental sciences. It is necessary to have people with original minds and an interest in industrial innovation, but the skills they need are essentially the same as those needed by software engineers or computer scientists generally.

In the past, most of the ground breaking research in computer technology was done at university research centers. This is not true now. In most cases computer science and software engineering educators do not fully understand the complexities involved with software engineering research (there are a few exceptions). Capers Jones [Jones91], a well know software researcher, had this observation about measurement implementation and research:

> Management consulting companies such as Software Productivity Research; DMR Group; Peat, Marwick & Mitchell; Nolan, Norton & Company; and Ernst & Young have often been more effective than universities both in using metrics and in transferring the technologies of measurement throughout their client base.

According to Daniel Berry [Berry92]:

> Software engineering research is necessary because software production is hard, much harder than many people seem to appreciate. Some generalize from the kinds of programs developed for completely specified classroom assignments, which cannot take more than a semester to complete and which are never run after they are handed in, to the belief that all software is

straightforward, is only few dozen pages long, and is just a matter of implementing the obvious, complete requirements of a single-person customer. Some generalize from the kinds of programs that are formally specifiable and whose compliance to these specifications is formally verifiable to the belief that all software systems are formally specifiable and verifiable. However, the fact is that real software developed to solve real problems is several orders of magnitude more difficult than the above toy problems.

Most research deals with simple problems, that is, the "real" problem's complexities are reduced to perform "valid" experiments on the problem area. In software engineering, we cannot reduce the complexities for two reasons. First, they are interrelated, and second, the purpose of the research is to develop methods to manage the complexities. Another reason we cannot reduce the complexities involved with software research is the inherent complexities within the problem area itself. The complexity of software research is the most complex research field known to man. According to Meir Lehman [Lehman91], there are three classifications of programs, which he calls the SPE scheme:

An *S-type* program is one required to satisfy some *pre-stated specification.* This specification is the sole, complete and definitive determinant of program properties.... In their context *correctness* has an absolute meaning. It is a specific relationship between specification and program.

A *P-type* program is one required to produce an acceptable *solution* of some ... problem. If a complete and unambiguous statement of that problem has been provided it may serve as the basis for a formal specification.... Nevertheless, program correctness relative to that specification is, at best, a means to the *real end,* achievement of a satisfactory solution.... In any event, when *acceptability* of the solution rather than a relationship of correctness relative to a specification is of concern the program is classed as of type **P**....

An *E-type* program is one required to solve a problem or implement an application in some *real world* domain. All consequences of execution, as, for

> example, the information conveyed to human observers and the behavior induced in attached or controlled artifacts, together determine the acceptability of the program, the value and level of satisfaction it yields. Note that *correctness* cannot be included amongst the criteria.... Moreover, once installed, the system and its software become part of the application domain. Together, they comprise a feedback system that cannot, in general, be precisely or completely known or represented. *It is the detailed behavior under operational conditions that is of concern.*

The E-type programs are the software engineering researcher's problem area. This problem area is similar to forecasting weather. The only time weather forecasters are correct is when they look out the door. Accurate short and long range weather forecasting is impossible. There are two many unknown and known variables to deal with. Super computers have choked on the data while running forecasting routines. Software engineering research and forecasting has the same problem. Daniel Berry [Berry92] commented on the Lehman's SPE system and how it relates to software engineering research:

> Software engineering research, when it develops tools and environments, is developing mainly E-type software. Initially, the software engineers, doing their own software development, perceive a need for a tool or an environment to do some of the more clerical parts of their task, to manage the software through its entire life cycle. From this perceived need comes a vague idea of the functionality of the tool or environment. However, this idea cannot be made less vague until the tool or environment is actually used. Thus, the software engineer builds a near-production quality prototype and tries it out, perhaps in the construction of the next version. Thus, the tool or environment has become an inextricable part of its own and other software life cycles.
>
> It is these impossible-to-specify and thus impossible-to-verify programs that are the subject of software artifact engineering research, and it is the

> regularization of the process of producing these artifacts that is the subject of software engineering methodology research.
>
> Perhaps, here we see the basis for the theoretician's condemnation of software engineering research. The kinds of programs they work with are S type. Perhaps they are not even aware of the existence of P-type and E-type programs, and they believe that all programs are S type or are easily made S type. If all one knows about are S-type programs, then it is quite reasonable to doubt the necessity of methodological research; it suffices to formalize the problem and the program rolls right out of the formalization. Certainly, S-type programs can be implemented quite systematically every time, and there is no need for software project management, for example, to build them. It is for E-type programs that the most help is needed; and if a technology, method, or technique is judged as useful, it is because it makes the production of E-type programs more systematic and repeatable.

Another problem the software engineering researcher faces is the appropriate number of subjects (population) to select for the research project. Capers Jones did not discuss why universities are falling behind the research power curve. The primary reason is that university students do not represent the software development environment. Stephen Schach [Schach93], a software engineering educator and researcher made this comment:

> When performing experimentation-in-the-many, students cannot be used as subjects.The difference of scale between classroom projects and real-life projects precludes the use of students in experimentation-in-the-small. The situation is considerably worse with regard to experimentation-in-the-many. The largest practical classroom group project is generally a team of three students working together for 10 weeks; when teams are larger, the actual work is usually done by only two or three members of the team. Bearing in mind that an undergraduate can probably devote at most 10 hours a week to a single course, this means that a team of three students can put in at most 2

person-months of effort. However, a project that can be completed in 2 person-months can hardly be considered to be programming-in-the-many. A larger effort is likely to be possible only from a graduate class, and even then 3 person-months is probably the upper limit. Furthermore, in order to be able to make valid statistical inferences, a minimum of 20 teams is needed. This means that the class size must be at least 60, which is not common at the graduate level in computer science. The experiment of Boehm, Gray, and Seewaldt to compare rapid prototyping with specifying was run with only seven teams, four of size 3 and three of size 2, a total of 18 graduate students. The average team effort was 2.7 person-months. There are a number of reasons why this experiment has been attacked, including the argument that the product was hardly large enough to constitute programming-in-the-many and that the subjects were computer science graduate students and not computer professionals. In addition, the experiment has not been repeated by an independent group.

Another problem with Software Engineering research is validating the results using statistical analysis. The problem is that the sampling population is not large enough to effectively validate the results. Stephen Schach discussed an experiment-in-the-small using about 50 students. A software engineer researcher wanted to validate a new technique for coding products that could result in fewer faults. The half of the 50 students are chosen at random to use the new technique, the other 25 are to use the techniques taught in class. These two groups were called Team A and B. Team A used the researcher's method and Team B used the traditional development methods. Stephen Schach [Schach93] expands upon this problem by taking the previously described experiment and creates an experiment-in-the-many by using software practitioners in the field . He also discusses Sackman's research on software practitioner performance:

> Suppose that team A completes the product in only 20 person-months, while team B takes 29 person-months. At first sight this seems to mean that the new design method used by team A promotes faster product development than the

> existing design method used by team B. But that is not necessarily the case. It is quite conceivable that the members of team A are simply better software engineers than those of team B and that the observed differences are due solely to differences between individuals in the two teams.
>
> Is it conceivable that such large differences can exist between individual software engineers? Sackman performed a series of experiments comparing the abilities of computer professionals. He observed differences of up to 28 to 1 between pairs of programmers with respect to items such as coding and debugging time, and product size. Superficially, this is easy to explain: An experienced programmer will almost always outperform an entry-level programmer. But that is not what Sackman measured. He worked with matched pairs of computer professionals, comparing, say, two individuals with 12 years of experience implementing operating systems in assembler. Another pair of his subjects might be two entry-level programmers, both trained at the same institute and both with only 2 months of programming experience in implementing data capture products. What is most alarming about Sackman's results is that his biggest observed differences, such as the figure of 28 to 1 quoted previously were between pairs of experienced programmers. Thus the difference between the times needed for the two teams [A and B], namely 20 person-months for team A and 29 person-months for team B could easily be explained by differences in the abilities of individual team members.

This is another reason why software engineering research is so difficult and so exciting.

In academia, publications are critical to the peer evaluation and academic promotion process. I have found that it is almost impossible to publish validated software engineering research because by the time you have validated what you have done, the technology is no longer current. I can write about what I do in the classroom, but to really get into the real research is more difficult. It requires a lot of time, money, and faith by the institution providing the funds and the institution acting as the research subject. Daniel Berry [Berry92] also commented about this subject:

> I have heard comparisons of published papers in conferences and journals of theoretical computer science and published papers in conferences and journals of software engineering. The claim is made that the theoretical papers involve much more work to bring to final published form than do the software engineering papers. This can be substantiated partially by the longer delays between submission and appearance for the theoretical papers. In addition, it takes much more work to get the first submitted version written. This observation may be true, but it misses part of the point. The way theoreticians work, the paper is the whole work. When an idea comes to the theoretician, he or she begins writing a paper. The development of the theory is the writing of the paper. On the other hand, even before a paper in software engineering can be written about a particular tool, environment, or software artifact, the tool, environment, artifact must be implemented, installed, tested, and used. Even before a paper can be written about experiences using a software method, management technique, tool, environment, or program, the tool, environment, or program must be implemented, installed, and tested; the users must be trained in the method, technique, tool, environment, or program; they must be left to apply the method, tool, environment, or program; and finally the authors must decide what has been learned. If one counts the work that must be completed before writing can be started, it is doubtful that the theoretician is spending more time to produce a paper than is the software engineer. Indeed, the labor intensiveness of software engineering research is the reason that the publication list of a good software engineering academic will not be anywhere near as long as that of a good theoretician.

Nature of Software Engineering Courses

What is the nature of our software engineering programs? In a word, dismall. Software engineering measurement undergraduate and graduate courses do not exist. Remember, this is the most important topic of software engineering, processes management, and research. According to Caper Jones [Jones91]:

> Academic institutions and universities are distressingly far behind the state of the art in both intellectual understanding of modern software measurements and the actual usage of such measurements in building their own software. The first college textbook on function points, Dreger's text on *Function Point Analysis,* was not published until 1989, a full 10 years after the metric was placed in the public domain by IBM. Even so, the author is employed by Boeing and is only a part-time faculty member. The number of major U.S. universities and business schools that teach software measurement and estimation concepts appears to be minuscule, and for the few that do the course materials appear to be many years out of date. The same lag can be observed in England and Europe. Interestingly, both New Zealand and Australia may be ahead of the United States in teaching software measurement concepts at the university level.

In addition to the lack of measurement courses, the computer programming assignments do not come close to replicating the realities of software development. This is affirmed by Daniel Berry [Berry92]:

> In addition, classroom exercises are woefully unrealistic in terms of the quality assurance and maintenance activities they require. Class programs are tested only enough to make sure that they will pass the grading test. They are forgotten once they are handed in, never to be maintained. It is well known that testing and maintenance account for about 60% of the cost of program production [Boehm81]. These two activities are difficult precisely because they involve the paths of interaction between parts. When tracking down the source of a bug, which usually shows up nowhere near the source, all possible

> paths of interaction must be followed backward from where the bug is observed. Moreover, each time a change is made, all possible impacts of that change must be explored. Because these interactions grow on an exponential scale, human creativity becomes an absolute necessity to cut through the combinatorial explosion to focus on the most likely places of interaction. Thus, the classroom programming exercises simply do not show the full scale of intellectual difficulty involved in software production. Those who generalize from software developed in the classroom come to unrealistic conclusions.
>
> The funny thing is that even these classroom-style exercises are harder than people think. There are several cases of authors promoting a certain systematic or formal way of working in a published paper containing a smallish, classroom-style toy example, only to end up red-faced as readers found and published corrections to their supposedly correct example.
>
> If classroom-style, relatively trivial exercises are so difficult to do right, what hope is there for any real-life, industrial strength or at-the-frontier program to be done right? Programs are complex animals, and the study of methods to manage that complexity is an intellectual challenge even greater than that of programming and mathematics, which are only tools of the process.

After our students receive their degrees, we hire them without seeing their work. Since 1988, I have encouraged my students to create a computer science or software engineering project portfolio. This portfolio could be a simple notebook binder or disk storage. The portfolio contains all their written work plus their best project work. Since National University was not well-known, I felt that the portfolio would give them an advantage when competing with Stanford, the University of California, and other prestigious computer science schools.

While researching material on quality management and productivity I discovered that Tom DeMarco met a computer science educator in western Canada that felt the same as I did. This is Tom DeMarco's [DeMarco87] story:

In the spring of 1979, while teaching together in western Canada, we got a call from a computer science professor at the local technical college. He proposed to stop by our hotel after class one evening and buy us beers in exchange for ideas. That's the kind of offer we seldom turn down. What we learned from him that evening was almost certainly worth more than whatever he learned from us.

The teacher was candid about what he needed to be judged a success in his work: He needed his students to get good job offers and lots of them. "A Harvard diploma is worth something in and of itself, but our diploma isn't worth squat. If this year's graduates don't get hired fast, there are no students next year and I'm out of a job." So he had developed a formula to make his graduates optimally attractive to the job market. Of course he taught them modern techniques for system construction, including structured analysis and design, data-driven design, information hiding, structured coding, walkthroughs, and metrics. He also had them work on real applications for nearby companies and agencies. But the centerpiece of his formula was the portfolio that all students put together to show samples of their work.

He described how his students had been coached to show off their portfolios as part of each interview:

> "I've brought along some samples of the kind of work I do. Here, for instance, is a subroutine in Pascal from one project and a set of COBOL paragraphs from another. As you can see in this portion, we use the loop-with-exit extension advocated by Knuth, but aside from that, it's pure structured code, pretty much the sort of thing that your company standard calls for. And here is the design that this code was written from The hierarchies and coupling analysis use Myers' notation. I designed all of this particular subsystem, and this one little section where we used some Orr methods because the data structure really imposed itself on the process

> structure. And these are the leveled data flow diagrams that make up the guts of our specification, and the associated data dictionary ..."
>
> In the years since, we've often heard more about that obscure technical college and those portfolios. We've met recruiters from as far away as Triangle Park, North Carolina, and Tampa, Florida, who regularly converge upon that distant Canadian campus for a shot at its graduates.
>
> Of course, this was a clever scheme of the professor's to give added allure to his graduates, but what struck us most that evening was the report that interviewers were always surprised by the portfolios. That meant they weren't regularly requiring all candidates to arrive with portfolios. Yet why not? What could be more sensible than asking each candidate to bring along some samples of work to the interview?

The portfolio is another measurement process. It provides an historical reference for the students. The portfolio also helps them to see their improvement, recognize their capability, and, in most cases, know their potential. The portfolio is one of the most important instructional tools available to the instructor.

Curriculum Analysis

I designed several computer science and software engineering courses as part of my Ph.D. internship. These courses vastly improved National University's Computer Science and Software Engineering degree programs.

I also performed an extensive analysis of National University's computer science and software engineering program in 1993. I have included my findings as an example of the problems the computer science and software engineering programs are facing today.

> Over the past year most of us have been extremely interested in modifying the B.S. Computer Science (BSCS) program. I think that we have done a fairly good job in defining new needs and justifying those needs with new and modified courses in this program. Most of the changes are productive and will

provide the student with an excellent foundation of theory and programming skills. Our graduates will serve the local software industry well, with some degree of additional education and training

Several questions that I cannot seem to answer are

- Is this a traditional science based computer science degree?
- Does this degree contain the necessary courses for a computer scientist?
- Are we providing the courses that are "really" required by our students?
- Are we providing the local and regional software industry with *bona fide* computer scientists or some hybrid that is not really a computer scientist but not really an information scientist either.

BSCS Problems

Over the past four years the [Computer Science project] instructors encouraged the students to develop their own ideas for a project. The instructors either accepted the project ideas or gave them another chance to come up with another idea. In most cases the first idea was accepted.

I have seen several problems with this approach.

1. The students become both the users and designers. A situation that is not normally seen in the real-world.
2. The students do not have the academic training in software engineering concepts, including project management. Therefore, their software engineering skills are lacking for a software engineering based project.
3. The instructors usually do not have a full grasp of the complexities of the project. Risk management is not part of the engineering process. Problems that should have been recognized in the first stages are not even noticed

until the last stages of development. The instructor has to compromise the student derived requirements so that they can finish the project.

4. The students drive the development process rather than the instructor. This is due in part to the students deciding on the project, the goals of the project, and finally the software requirements of the project. The instructor plays a minor role in this decision process.

5. The software engineering concepts of analysis, design, testing, and implementation are not fully followed in the project. The students start coding before they actually look at the project's software requirements. They do the documentation to satisfy the requirements of the course work, not to satisfy the software engineering requirements. In other words, they do not gain the benefits of actual software engineering due to their inexperience with the process. Their primary objective is to complete the course requirements in any way possible, not the software engineering process.

6. The students expect a high grade due to the number of hours spent on the project development. The quality of the work doesn't matter. The mechanisms to track each student's part in the software development is either partially recognized or missing entirely. The student's grade is based almost entirely on the teams efforts not their efforts. This provides an environment for hard working students to carry the team well beyond the team's capability or for the team to carry a weak student thus allowing a few team members to do most of the work.

7. The instructors are placed into a role of project manager or division manager. They know what the project is suppose to do but not know how to construct the project. Again, the project complexity comes into play, whereby the instructor is caught off guard if some unforeseen problem arises.

8. The students are encouraged to use software development tools during the project. However, as a whole the students are not trained on specific tools, such as code generators (which should not be used during the project courses), screen generators, graphic user interface development environments, etc. Generally, only one team member is experienced with a specific tool and tries to train the other team members to use this tool. This creates several additional problems. During the first two months the students are learning the software engineering concepts (which encourages the use of tools) and the various development tools available for rapid prototyping and development. There is very limited time for the students to learn the software engineering concepts and internalize them and at the same time learn new software development tools and become proficient with them within four or five weeks. This creates a very stressful situation for the students and the project instructors.

Solutions

How can we eliminate most of the [Computer Science project] problems and yet at the same time provide a creative environment for the CS project students? First we need to get out of the mind set that we must provide a open ended development environment for our BSCS students. If we are going to create a software development environment similar to the real-world, we must comply with the constraints of the real-world. As such we must provide the following constraints:

1. Provide coding standards that are consistent to the recommended industry standards. These standards must provide the foundation for structured programming concepts. The standards should be general for the programming languages used at National University (Pascal, C, C++, and Ada) and at the same time provide standards for specific language characteristics.

2. Provide document standards that are consistent, in some degree, with the recommended industry standards (IEEE and DOD).

3. Limit the software development tools used by the students to those provided by the National University. We must actively promote the use of these tools of the students by the faculty. I believe that we have a very good software development environment that fits into the scope of what we are trying to teach in our academic environment. Let's not forget that our goal is to provide industry with students who know how best to use the resources provided, not students who will be at a loss if they do not have specialized software development tools to create their source code or menu system.

4. Provide a tool that will encourage the same level of work from all the students in the course. This tool is the *Weekly Team Status Summary Report.*

5. Grading criteria must be set up to allow for individual work as well as team interaction.

6. Provide a CS project information and policy form for each student that will define the following:

   - Project agreement description

     Project proposal

     Project criteria

     Student, Team, and Instructor responsibilities

     Ownership of project copyright

     Project deliverables

     Suggested projects

   - Description of the first assignment

- Description of the weekly team status reports

7. A Computer Science / Software Engineering (CS/SE) Quality Management Team should be formed that will provide both assistance and quality control of the project classes. The responsibilities of the CS/SE Quality Management Team are as follows:

   - Review the CS/SE curricula annually and make recommendations to San Diego for review.
   - Modify the CS/SE curricula to meet the needs of the local community. The curriculum foundation as required by the National University CS/SE department will not be changed.
   - Determine problem areas in curriculum continuity between courses and make recommendations for improvement.
   - Review textbooks that may fit the local industry needs and make recommendations to San Diego for acceptance.
   - Review the Student Work Experience database and make recommendations to the CS/SE project faculty for project team structure.
   - Work with the local industry to determine specific areas of instruction that the industry would like National University to offer to their students, especially if they are paying the tuition.
   - Seek out software projects from the local industry for both the BSCS and M.S. Software Engineering (MSSE) project courses.
   - Write system requirements and software requirements for the BSCS project courses based on the needs of the local industry software projects. The BSCS project courses are limited to design and implementation only.

- Write statement of work for the MSSE project courses based on the needs of the local industry software projects.
- Help the BSCS and MSSE project faculty with research when the project teams are doing research out of the project faculty member's expertise.
- Review all the material created by the project course students and make student grading recommendations to the project faculty.
- Review all CS/SE course outlines to determine if the proposed course material contains the basic requirements of the National University CS/SE department and the recommended material of the Sacramento Regional Academic Center Department of Computer and Information Science Quality Management team. If needed, make changes in the course outlines and notify the appropriated instructor of the changes and the reasons they were made.
- Perform quarterly reviews of the CS/SE students' performances to determine if any student is at risk and may need remedial training or is a possible candidate as a tutor.
- Semi-annually perform a self assessment to determine areas of improvement in management, interaction with the CS/SE faculty, and how to improve student performance.
- The CS/SE Quality Management team members will perform the above tasks continually and meet once a quarter to discuss their findings and assessments.

8. A work experience evaluation form should be given to the students at different points in their program. These points occur when they start their first programming language course, the first systems course, and when the Project Agreement and Policies document is given to the students. The evaluation form will help the instructors by providing

- student team leaders, those student with excessive work experience, and
- information to assign students equitably to a project team.

9. The CS project instructor, along with the CS/SE Quality Management Team, should decide which students will be assigned to a specific project. This will eliminate the problem of having all strong students on one team and all weak students on another team. It will also provide a real-world environment, such as, that a company project manager decides which employee will work on a specific company project.
10. The instructor or a member of the CS/SE Quality Management Team will become a user for the project teams. The project selection committee will assign specific projects to the CS project course. These projects may be academic in nature or projects required by the local business community. The students are not allowed to select their own projects unless it is for their company and their company is willing to become actively involved with the project.
11. Provide a small seminar on how to deal with personal conflict problems. This seminar should help eliminate the dominate team member's influence in driving the team direction as well as the timid team member, who may have great ideas, but is intimated by the dominate team member.

M.S. Software Engineering

National University's M.S. Software Engineer program is one of the best in the United States, if not the world. We have looked at the industry needs and other academic programs back in 1988 and modified the MSSE program at that time to meet the needs of the software industry. We all felt that we really did a superb job defining a curricula that met the needs of the local and regional industry.

Five years have passed, and we have not kept abreast with the rapid advances in software development tools, software project management systems and methods, software quality management systems and methods, risk management issues, and the software industry needs. To bring the scope of the MSSE program back into an actual software engineering program we must modify and add several MSSE courses. This will prevent the MSSE program from becoming a graduate computer science program with a software engineering emphasis.

We do not have any undergraduate foundation courses for the software engineering module courses. We do not have an "expert system" or AI undergraduate foundation course for CS 625. These course are introduction courses because the graduate students do not have the background or training necessary to take the appropriate advance subject material. We must have undergraduate courses to support the graduate courses; otherwise all we are doing is teaching advance undergraduate courses and giving credit for graduate work.

I intend to pose some serious questions about our MSSE curricula and intend to provide answers that are supported by the software industry and professional societies.

Problems

Since 1988, the MSSE program has been divided into 4 modules with each module independent of the other modules, except for the project module. The problem with this concept is that the modules do not reinforce the material covered in the other modules. This is especially true of the current design of the database course, which expects the students to design a database SQL front-end and database engine. How can we expect these students to come up to software engineering graduate level standards when they do not know the methodologies or techniques to analyze and design this project.

We have a problem with the system courses. These courses are designed as traditional "graduate" computer science courses. I have reviewed six California and one out-of-state undergraduate and graduate computer science programs. The syllabus description of our system courses falls directly in line with all seven of the upper division computer science course descriptions for essentially the same courses. We need to review these courses and bring them into line with software engineering concepts for hardware/software systems and transportability issues, advance operating systems and portability issues, advance networks and the use of them in software development.

We have a problem with the "traditional" software engineering courses, CS 620, CS 622, and CS 626. There are faculty who expound the benefits of object-oriented design while at the same time glossing over the benefits of structured analysis and design.

Six very complex concepts are delegated to CS 620, while only one complex subject is delegated to CS 622. The students have to learn (and if given enough time, practice) the following software engineering concepts in CS 620:

- Software project planning
- Feasibility studies
- Software requirements analysis
- Structured analysis
- Structured design
- Project plan document
- System requirements definition document
- Software requirements specification document
- Software detailed design document.

This course is taught as a traditional undergraduate "Introduction to Software Engineering" course without the verification and validation material. The students are not given enough time to digest the material and to internalize it.

The second software engineering course, CS 622, dedicates four weeks to software object-oriented design concepts and techniques. Again the students have to go through a review of object-oriented programming concepts. In most cases the students are not familiar with either object-oriented design or programming concepts. This is the main reason for the course being dedicated to this subject is because the students usually do not have the proper foundation to take this course and National University does not provide the proper undergraduate foundation courses for the MSSE graduate students.

The third software engineering course, CS 626, only covers a very small portion of the material that a software engineering professional must have, software quality management. CS 626 teaches the students verification and validation concepts and techniques, otherwise known as unit, integration, and system testing. It does not properly cover configuration management, software quality risk analysis, software quality factors, software complexity factors, and a host of other quality issues that must be covered in this course.

CS 623A-B is now taught as a database management system design course, requiring the students to build a structured query language (SQL) front-end and a database engine. In other words, this course is designed as a computer science graduate course. This course should be designed as an advance database design course.

Finally, the last course, CS 625, Expert Systems, should be removed from the program entirely. This course is a specialized computer science course and taught as a introduction undergraduate course since we do not have a undergraduate foundation course.

Over the next few pages I am going to attempt to discuss the above problems and give some viable solutions to them.

Solutions

CS 620 — This course should be taught as a "Software Requirements and Analysis" course. This course will use two small projects to help the students be able to understand

- the process model for requirements analysis
- the software life cycle requirements and analysis phase models
- software structured, real-time, and object-oriented analysis
- how to eliminate ambiguity in user-developer communication
- the purpose of the software requirements specification
- the difference between the different analysis methods (structured, real-time, and object-oriented)
- elimination of requirements based risk
- how and when they should use a particular method for their software product
- the Software Engineering Institute maturity model.

The will also learn how to develop detailed system specifications and software requirements specification documents.

The small projects developed in this course will be further developed in CS 622, the Software Quality Management course, and the Software Project Management course.

CS 622 — This course should be taught as a "Software Design Methods" course. The students will continue using the small projects in this course to translate the analysis and requirements into a software design. The students will learn

- the process models for software design
- software structured, real-time, and object-oriented design methods
- how to create and evaluate the software detailed design document
- how to transform and translate the structured analysis into a software structured design
- advance technical writing skills

They will see the difference between the different design methods (structured, real-time, and object-oriented) and learn how and when they should use a particular method for their software product.

CS 626 — This course should be changed to a "Software Quality Management: course. See CS 621A.

CS 624A — This course should teach the MSSE students on the problems, solutions, and methods, techniques required to integrate software and hardware together.

CS 624B — This course should teach the MSSE students on the different operating systems and how each can enhance or hamper software development. It should also teach

- the problems and solutions to port software from one operating system to another, and
- address the issues and problems in developing special purpose operating systems (for example, embedded real-time systems, object-oriented systems).

The students should do a research project on how an application written for one operating system can be ported to another operating system. The students should look at the application's use of operating system dependent code and how much effort would be required to modify this code, the user interface, operating system efficiency, etc.

CS 624C — This course should be an advanced networks course dealing with software development using network technology.

CS 623A-B — This course should provide a through understanding of database design. The students should learn how to design, develop, implement, and evaluate advance database systems for micros, minis, and mainframes. The course should focus on exploring the architecture and functionality of operational object-oriented databases, understanding object-oriented data models, designing a distributed database architecture, and understanding client/server technology. This course should use a small project to help the students be able to understand

- the object-oriented paradigm
- client/server and distributed technology
- the relational model and its DBMS implementation
- entity-relationship and object-oriented data modeling
- advance normalization
- the different hardware and software implementations
- Structured Query Language
- embedded SQL in high level languages
- DBMS security concepts
- DBMS integrity concepts
- relational database data definition concepts
- relational database data manipulation concepts
- data distribution methods
- distributed transactions

CS 625A — This course should be eliminated. It is an introduction course to expert systems.

CS 627A-C (MSSE Project) — The basic definition of these courses should not be changed. The students in this course should developed a project for the local community or an instructor directed project. This will prevent them from becoming the user/developer. The students need to develop user-developer communication, and they cannot do this if they are the user. Also the MSSE students should work with the Management/Business students as they work with the local community businesses. This would give the MSSE and Management/Business students experience with technical and non-technical communications.

As for the proposed research project, the recommended CS/SE Quality Management Team will provide the academic support for the students as well as the CS 627 instructor. If the CS 627 instructor does not have the expertise to work with the student's research project, then the CS/SE Quality Management Team acts as the user.

If we are going to create a software development similar to the real-world we must comply with the constraints of the real-world. As such we must provide the following constraints as described for the BSCS program with the following addition:

1. Provide extended time during the CS 627A course to develop the expertise necessary for specialized software development tools. We must actively promote the use of these tools of the students by the faculty. I believe that we have a very good software development environment that fits into the scope of what we are trying to teach in our academic environment. We can extend CS 627A by expanding the course to a two month format.

Recommended MSSE Module Sequence

Right now the MSSE program is made up of four three-course modules. Each module is independent of the other, except for the three course project module. If we add the Software Quality Management and Software Project Management courses and drop the Expert System and Verification and Validation courses, the MSSE module makeup would be as follows:

Advance Software Engineering Module (Module 1)

CS 620 - Principles of Software Engineering (Software Requirements and Analysis)

CS 622 - Advanced Software Engineering (Software Design Methods)

CS 621A - Software Quality Management

CS 621B - Software Project Management

Advance Database/Client/Server Module (Module 2)

CS 623A - Database Management I (relational systems)

CS 623B - Database Management II (object-oriented systems)

Software/Hardware Systems Module (Module 3)

CS 624A - Principles of Hardware and Software Integration

CS 624B - System Software

CS 624C - Networked Computing Systems

Advance Software Project Module (Module 4)

CS 627A_1 - Software Engineering Project Ia

CS 627A_2 - Software Engineering Project Ib

CS 627B - Software Engineering Project II

CS 627C - Software Engineering Project III

Conclusion

Overall, I think that we have done an extremely good job designing courses for the B.S. Computer Science and M.S. Software Engineering programs. We need to keep up with the rapid advances of hardware and software and incorporate these advances into our course material and labs as much as possible. This will require a commitment from both the faculty and administration.

We must redefine the goals of our B.S. Computer Science program. We must ask ourselves:

- Are we providing the courses required by the students and their employers or potential employers?
- Is this a true computer science program, or is it a computer science program with a built in software engineering emphasis?
- If it is a software engineering emphasis, are we providing those courses necessary to develop junior software engineers?
- Are we providing the proper foundation courses for the M.S. Software Engineering program? If not, are we sure that the graduate courses, that require a foundation course, are actually providing advance topics to the students. I have talked with many of the MSSE instructors, and all of them tell me that they spend from 25% to 50% of the course reviewing introduction course material.

As for the M.S. Software Engineering courses, most of these courses are taught as post-graduate computer science courses. The primary reason stated for this, is that we must ensure that the MSSE students understand the implementation issues involved with the applications. First of all, why are we "ensuring" that the MSSE students understand the implementation issues by having them actually implement design?

If I am hired as a software engineer, my employer expects me to know how to design, determine the risks, guarantee the software quality, and get the project complete within time and under budget. As a software engineer I would hire a senior programmer to deal with the implementation issues. The senior programmer is not expected to have the expertise of the software engineer.

If some of the MSSE faculty are concerned that we must ensure that our MSSE students know how to implement computer science concepts, then we must provide this education within the B.S. Computer Science program, not the MSSE program.

Our primary goal for the MSSE program is to produce software engineering students that fully understand the entire software development process which includes how to bring a Software Engineering Institute Capability Maturity Model Level 1 company to a Level 4. If we can meet this goal, we will be the only university in the United States, possibly the world, to produce world class software engineers.

Conclusion

As a conclusion to this and the previous academic sections I offer a statement by Daniel Berry [Berry92]:

> It has been observed that computer science is the science of complexity. Nearly everything computer scientists work on is geared more or less to reducing or managing the complexity of some system, be it hardware, software, firmware, or people. Software is the most malleable of the wares that are the subject of computer science; its very malleability is a continual enticement to attempt more and more ambitious projects that are beyond what can be done by special-purpose hardware and firmware and what can be done by people. The ambition leads to attempting more and more complex tasks for which the only hoped for solution lies in reducing and managing that complexity.
>
> Managing software complexity demands a deep understanding of software. It

also demands a good understanding of hardware and firmware. Because software is created by people and groups of people, managing software complexity demands also a good understanding of people and groups, and that understanding pulls in elements of psychology, sociology, and management. Moreover, if someone claims that software engineering is no more than psychology, sociology, and management, simply ask this person if he or she would want the air traffic controller software that lands his or her next flight to have been written by a psychologist, sociologist, or manager who does not also have a deep understanding of software in particular and computer systems in general. Can you, the reader, imagine how someone without an understanding of how a tiny change to a program can cascade into dozens of seemingly unrelated bugs, of how algorithms can have different orders of complexity, and of what abstraction tools and concepts have been developed to contain complexity can possibly be relied upon to produce quality software for critical applications on which all of our lives depend?

Software engineering is intellectually deep and is a vital area of academic study. People who engage in this study should be afforded the same academic respect that is given to other, more established disciplines.

Sample Academic Software Engineering Products

I have included several software engineering document products that illustrate the complexity of software planning, analysis (requirements), design, and testing within the *Software Engineering Academic Project Management Production Tools* software application. You can access these samples by selecting an "example document" located in the *Documents* tab section. The following document examples can be either viewed and/or printed using Microsoft Word for Windows, version 6.0:

1. Software Project Plan without Gantt charts (B.S. Computer Science project)
2. Software Requirements Specification (B.S. Computer Science project)

3. Software Detailed Design (B.S. Computer Science student project example)
4. Software Users' Manual (B.S. Computer Science student project example)
5. Software Test Plan and Procedures (Software industry project)

2. The Software Development Process

This section discusses the software engineering methodologies used to develop the Software Engineering Academic Project Management Production Tools (C-ProMPT) software application. The first section discusses why I designed and built the application, and the remaining sections discuss the methods used to analyze, design, construct, and test the application.

Why C-ProMPT?

Academia needs an inexpensive instruction and tool set software application that provided consistent software development information , standards, and guidelines for Computer Science and Software Engineering students throughout their academic program. According to Watts Humphrey [Humphry95]:

> Software is now a critical element in many businesses, but all too often the work is late, over budget, or of poor quality. Society is now far too dependent on software products for us to continue with the craft-like practices of the past. It needs engineers who consistently use effective disciplines. For this to happen, they must be taught these disciplines and have an opportunity to practice and perfect them during their formal educations.
>
> Today, when students start to program, they generally begin by learning a programming language. They practice on toy problems and develop the personal skills and techniques to deal with issues at this toy problem level. As they take more courses, they improve these methods and soon find they can develop fairly large programs relatively quickly. These programming-in-the-small skills, however, are inherently limited. While they may have sufficed on small-scale individual tasks, they do not provide an adequate foundation for solving the problems of large-scale multiperson projects.

> This [software application] follows a fundamentally different strategy. It scales down industrial software practices to fit the needs of small scale program development. It then walks you through a progressive sequence of software processes that provide a sound foundation for large scale software development. Once you have learned and used these practices on small programs, you will have a solid foundation on which to build a personal software engineering discipline.
>
> The principal goal of [C-ProMPT] is to guide you in developing the personal software engineering skills that you will need for large-scale software work.

History

I started teaching graduate Software Engineering courses in 1988. Right away I recognized that the students lacked discipline in all aspects of software development. To add some degree of discipline to the students' software engineering academic environment, I developed a set of document and coding standards. The standards improved the software development environment, however, a lot of work was still needed to improve the academic software development processes.

The document and coding guidelines and standards have undergone almost a dozen changes since I started using them in my courses. The initial set of document standards were a subset of the Institute of Electrical and Electronics Engineers (IEEE) and Department of Defense (DOD) document standards. Both the document and coding standards were revised at least twice a year at the end of each computer science and software engineering project course. The project team members would make change recommendations. If the recommendations were valid, I would change the standards.

About a year ago, I realized that the standards were simplified down to "toy" standards. They were not challenging for either the undergraduate senior level and graduate students. The students were missing an important software development component, negotiating with management to modify the project team's compliance with the organization's standards. I reviewed my standards and the new IEEE and DOD Mil-Std-498 standards. The new IEEE and DOD standards made sense; they were simpler (especially the DOD standards) and provided several ways to document the newer software engineering analysis and design methods. My standards were modified to reflect the IEEE and DOD standards changes.

The C-ProMPT document standards are more complex. However, the students are encouraged to review the standards and chose the components that they need to manage their project. The students have to justify why they have to deviate from the standards. This process provides another software development learning experience for them.

The coding standards were not changed very much over the past three years. The students and faculty recommended that I clarify several of the standards and guidelines and add a couple of guidelines, for example, commenting guidelines.

When I started teaching at National University the computer science and software engineering students were not required to take technical report writing. Most of the students did not know how to write technical documents. None of the students knew anything about document design concepts, that is, how much white space to use, when to use bold and italics for emphasis, the appropriate use of graphics, etc. The students' information in their documents were fine, but their document esthetics were horrid. The bottom-line was that their documentation was not enjoyable to read.

To change this weakness, I developed the Document Design Fundamentals. The design fundamentals covered the essential document esthetics. This subject became part of my document standards lecture. Right after I introduced the design fundamentals the students documentation esthetics improved tremendously. Their documentation was easy to read and assess.

In 1993, I started using Word for Windows instead of WordPerfect. One of the unique features of Word was the ability to write Windows based software applications within the Word document. In other words, the document could be automated for specific tasks. After spending about six months experimenting with Word and WordBasic (Word for Windows programming language), I developed the software engineering document templates. These templates were based on the C-ProMPT software engineering document standards and incorporate the document esthetics as recommended in the Document Design Fundamentals. The templates allowed the students to concentrate on the technical material instead of the document esthetics.

Right after I developed the Word for Windows document templates, a student asked if I had all the guidelines, standards, document design fundamentals on-line. At that time, the entire suite of guidelines and standards barely fit into a 3-inch binder. The students were required to bring their copy of the suite to class. Thanks to the student's question, C-ProMPT was born.

Requirements

Software

The requirements for this system were fairly simple. The system had to provide the following:

Software Engineering Document Standards

Proven academic software engineering standards based on IEEE and MIL-Std-498 standards. The following standards were required for the C-ProMPT software application:

- Software Development Plan
- Software Project Plan
- Software Requirements Specification (multi-variant)
- Preliminary Software Detailed Design (multi-variant)
- Software Detailed Design (functional)
- Software Detailed Design (object-oriented)

- Software Test Plan and Test Procedures
- User's Manual
- Software Test Case Specification
- Test Case Incident Report
- Test Case Log Report
- Test Summary Report

The standards were based on the work of Gregory Russell, Brad Bowes, Chris Kolonis, and Dan Osier, National University, Sacramento, Computer Science / Software Engineering Quality Management Team

C and C++ Programming Language Coding Standards

Standards provide consistency and are specifications for a preferred development method. They also provide a framework for greater creativity. The C-ProMPT software application shall provide at a minimum C coding standards that can be used for C++ programmers. The standards shall use proven academic programming standards based on common software industry practices and guidelines. C-ProMPT shall provide the following coding standards:

1. General coding standards
    - suggested use of comments
    - code reviews
    - simplicity of design and implementation
2. Files
    - file header format
    - file identification format
    - file modification format

- file naming conventions
- size of source files

3. Functions
    - function header format
    - function prototypes
    - lexical rules for functions
    - methods of coupling modules together
    - placement of functions in the source file
    - place of main() in the source file
    - suggested size of functions
4. Data and Variables
    - choosing variable names
    - lexical rules for variables
    - maintainability of constants
5. Operators

   character tests

   dependencies on evaluation order

   lexical rules for operators

   order of side effects

   parenthesis
6. Control Statements
    - lexical rules for control structures

Document Design Guide

I have found that most of our students do not have any training in document design. C-ProMPT shall provide a document design help system that will provide the user with basic document design, layout, and typography. The guidelines shall use the current software industry standards (Hewlett-Packard, Intel, or Sun Microsystems) but modified for academic use.

Automated Software Engineering Document Templates

C-ProMPT shall include Microsoft Word for Windows document templates that are based on the C-ProMPT software engineering required document standards. The templates shall guide the technical writer through the documentation composition eliminating errors and preventing incorrect formatting. The templates shall reduce the documentation composition time by a minimum of 20%.

The following software engineering documentation templates required for the C-ProMPT software application:

- Software Development Plan
- Software Project Plan
- Software Requirements Specification (multi-variant)
- Preliminary Software Detailed Design (multi-variant)
- Software Detailed Design (functional)
- Software Detailed Design (object-oriented)
- Software Test Plan and Test Procedures
- User's Manual

At a minimum the templates shall provide the following features:

1. Automated input for:

- Project name
- Developer's company name, address, city, zip code, voice phone, fax phone
- Client's name
- Developer or team member names
- Client representative's name
- Project manager's name
- Quality assurance manager's name
- Project leader's name

2. Automatic caption for figures and tables
3. Automatic table of contents, list of figures, and list of tables generation and update
4. Generic text for standard paragraphs
5. Title page, record of changes, copy-right notice, and appropriate headers
6. Automatic insertion of iterative sections, design document only (for example, multiple objects and methods)
7. On-line help for each template feature
8. Linkage to C-ProMPT help system

Software Estimation Tool

C-ProMPT shall provide a documentation size and cost estimation tool. The documentation estimation tool is based on my senior technical writer experience and research.

User Interface

The C-ProMPT application shall use the interactive design principles as describe in Shneiderman92 and Hix93:

1. **Strive for consistency.**

    a. Sequence of actions shall be consistent from screen to screen.

    b. Prompts, menus, and help screens shall use identical terminology.

2. **Enable frequent users to use shortcuts.**

    a. If appropriate, abbreviations, special keys, hidden commands, and macro facilities shall be used to reduce user response times

3. **Offer informative feedback.**

    a. The system shall provide visual feedback for operations that take longer than 2 seconds. The feedback shall consist of a window indicating:

        - The time remaining for the operation for operations greater than 5 seconds

        - A "standby" message indicating that the operation will complete within 3 seconds.

4. **Offer simple error handling.**

    a. The system shall incorporate error handling routines that will prevent obvious user input errors. These routines will check user entries for appropriate data types; for example, an integer value is expected, the error routine will reject all entries except integers. An error message will inform the user of the desired input as well as what the user did wrong.

    b. The system will validate all user entries and database retrievals.

5. **Permit easy reversal of actions.**

    a. The user shall not have to retype the entire erroneous command but shall be allowed to edit the data entry. For example, if the user entered 1289e76 when he wanted to enter 1289376, the system shall allow the user to backspace to delete the "e" and enter "376."

    b. The system shall provide an Undo and Commit feature for all database entries. The "Undo" command will back the database to the point of the last "Commit" command. The "Commit" command will perform all user entries and actions perform since the last "Commit" command.

6. **Support internal LOCUS of control.**

    a. The system shall provide an atmosphere where the user has the impression that they are in command. For example, if the system is waiting for data entry, use this feedback message, "Ready for next data entry," rather than "Enter next data entry."

7. **Reduce short-term memory load.**

    a. The system shall have tabs that will inform the user of the location of each screen.

    b. The system shall have all "Help" command buttons located in the same area for each screen.

    c. The system shall have all "information blocks" located in the same area for each screen.

    d. The system shall use command buttons to access the C-ProMPT help topics.

8. **Organize the screen to manage complexity.**

a. The system shall limit major groups to a maximum of seven items. The major groups shall be easy to recognize by using either a color scheme or larger font sizes.

b. The system shall limit minor items within a group to a maximum of seven items.

Production Tools

The selection criteria for the production tools shall be:

1. Rapid prototyper tool for Microsoft Windows software application and on-line help development
2. Borland's Database Engine an integral part of the development tool
3. Uses a common programming language, for example, Ada, Basic, Pascal, C, or C++
4. Modularity for low coupling and high cohesion
5. Generates compact executable code without the use of external support files
6. Converts Microsoft Word documents to RTF files suitable for Microsoft Windows Help files.

Based on the above criteria, Borland's Delphi was selected as the software application development tool. Delphi is based on Borland's Object Pascal and uses similar constructs as Ada (specification and body). Dephi can use Paradox data files without additional programming. One of the features of Delphi is that you can build and access Paradox tables without having a licensed copy of Paradox, a plus. Delphi utilizes a compiler, so its code is very compact and usually in one executable file (unless you are using VBX extensions)

Microsoft Visual Basic was not selected because it used an interpreter and required several support files to run.

RoboHELP was selected as the best full-featured authoring tool for creating Windows Help systems and stand-alone hypertext information systems. RoboHELP greatly simplifies the process of creating and maintaining Windows Help files. It integrates fully with Microsoft Word, version 6.0a. It also provided custom Help buttons for Delphi which eliminated the need to program context sensitive Help routines for C-ProMPT. Another feature provided was a complete Help debugging tool.

Design

The primary criteria for the design was very simple, the C-ProMPT software application must be easy and fun to use. The secondary criteria was that it must provide support for the concepts learned in the computer science and software engineering courses.

Rapid Prototype Design and Development

I made the decision to develop C-ProMPT as a prototype software application. The prototype would demonstrate the design concept rather than a fully developed system. Jenny Preece [Preece93] defines a prototype system as:

> ... a software system that simulates or animates the structure, functionality, operations or representations of another system. A prototype should be cheap to produce and should take only a short time to develop.
>
> A prototype is a software system that:
>
> - actually works, that is, it is not an idea or drawing
> - will not have a generalized lifetime; at one end of the spectrum it may be thrown away immediately after use, at the other end it may eventually evolve into the final system
> - may serve many different purposes

- must be built quickly and cheaply
- is an integral part of an iterative process which also includes modification and evaluation.

A prototype will concentrate on some aspects of an interactive system and ignore others, and may differ from final systems in size, reliability, robustness, completeness, and construction materials.

- Full prototypes contain complete functionality but provide less performance than the final system.
- Horizontal prototypes demonstrate the operational aspects of a system but do not provide full functionality.
- Vertical prototypes contain full functionality but only for a restricted part of a system.

C-ProMPT was developed as a combination of a horizontal and vertical prototype system. The PSP Process is not fully functional, but the other C-ProMPT topics are fully functional.

Graphic User Interface Design

I wanted the graphic user interface (GUI) design to employ human-computer interactions concepts. These concepts are described by Jenny Preece [Preece93]:

> The user interface has a specific form of dialogue which is designed to facilitate user computer interaction. This dialogue enables the user to map (or relate) the details of tasks to the functionality of the computer system.
>
> A well-designed user interface makes it easy and natural for a user to break down (or decompose) a task into subtasks and map them on to the system's functions. A poorly-designed computer system requires its user to decompose tasks in unnatural ways, and the ensuing mapping is then prone to errors.

An important part of human-computer interaction (HCI) work, therefore, involves understanding the nature of users' tasks and the ways in which users most naturally decompose them. This, in turn, requires understanding the characteristics of users themselves and the influence on their behavior of the context in which they work. In addition, designers have to take into account technical and logistic considerations.

The goals of HCI are to develop and improve systems that include computers so that users can carry out their tasks:

- safely
- effectively
- efficiently, and
- enjoyably.

These aspects are collectively known as usability. A well designed computer systems with good usability can:

- improve the performance of the workforce
- improve the quality of life
- make the world a safer and more enjoyable place to live in.

The C-ProMPT GUI design incorporated most of the features described by Preece. In order to develop and integrate the screen designs, I needed to understand the following:

- cognitive psychology and organizational psychology

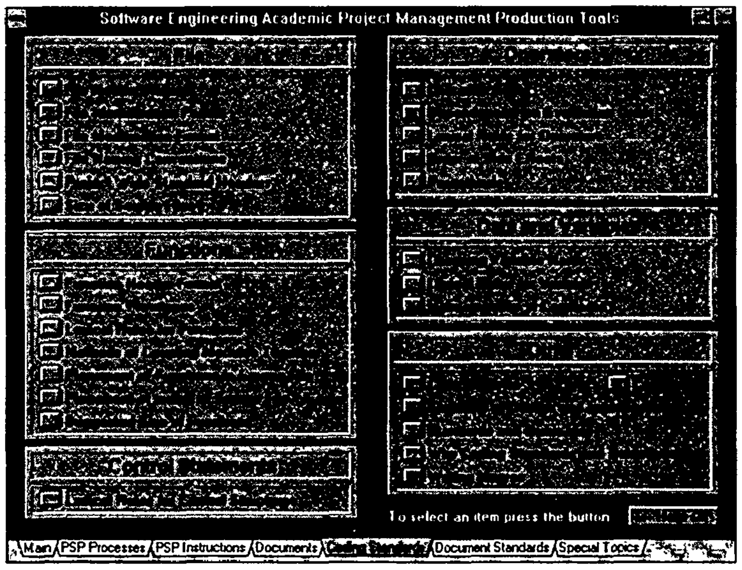
  Human-computer interaction is essentially cognitive, that is, it involves the processing of information in the mind. The overall aim of applying cognitive psychology to system design is to ensure that this information processing activity is within the capabilities of the users' mental processes.

In order to maximize the software application's use when introduced into the academic project environments it is also necessary to apply organizational and social psychology.

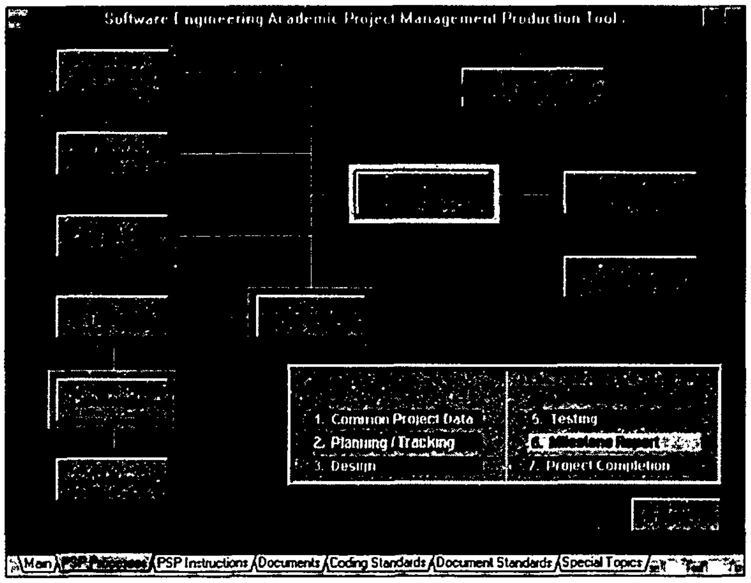
- established techniques for input, output and user support provision such as menus and forms, cursor control and on-line aids (industry GUI guidelines and standards)
- experience of other designs and knowledge of other systems

The entire C-ProMPT design dealt with cognitive psychology, some organizational and social psychology, GUI guidelines (IBM and Microsoft), and examples from other Windows based software applications.

Each GUI screen dealt with a specific topic and related sub-topics. Besides using related information in each screen, the user's learning ability was taken into consideration by designing an intuitive navigation system. Each screen has its major topics designated in bold fonts and within separate panels. The user automatically scans the major topics prior to scanning the sub-topics. For example the following screen contains six major topics and less than eight sub-topics per major topic.


Software Engineering Academic Project Management Production Tools
To select an item press the button
Main
PSP Processes
PSP Instructions
Documents
Document Standards
Special Topics

All of the C-ProMPT screens were designed in a similar manner, with some minor exceptions. All the screens' layout design is similar, except the *PSP Process* screen. This screen's layout provides a PSP process sequence graphical representation. However, the layout follows the same design concepts. The following screen shows the difference in style.



This screen has 12 push buttons and one "legend" panel. The push buttons were organized into six color-coded major topics. By using color-coded topics, the push buttons become sub-topics and maintain the layout design of the other C-ProMPT screens.

Help Screen Icons

The C-ProMPT help screen icon design also followed HCI design concepts. There are several advantages in using icons instead of command names, in that, in most cases, they are easier to learn and remember. They achieve this by:

- providing more visual information about the underlying object
- acting as powerful mnemonic cues

- explicitly showing the relationships between system objects

As I designed the Help Screen icons I had to take into account:

- The context in which each icon was used. This is because the context influences the comprehensibility of the icons.
- The task domain for which they are used.
- The graphic form that is used to depict the object.
- The nature of the underlying object being represented.
- The extent to which one icon can be discriminated from other icons displayed.

Since most of the "objects" were related in some manner, I decided to design the icons using a combination of graphics and text. This combination is useful in that the user will eventually associate the graphic symbol with the text.

Guidelines and standards

According to Jenny Preece [Preece93]:

> Guidelines occur in several forms:
>
> - High-level and universally applicable design principles need to be employed to direct the design and integrate ideas on design into a sound framework
> - Design rules are sometimes used to instruct a designer how to achieve a principled design that is appropriate for the particular system in question.
> - Systems should conform to international, national and industry standards.
>
> Ultimately, there are only good and bad design decisions, which reflect the way in which design guidelines are applied. Designers need to choose and apply the design guidelines intelligently at the right time. Attitude, experience, insight and common sense help in this process.

Guidelines

The term guidelines encompasses both the broad principles, which offer general advice and provide a sound foundation for a design, and the specific design rules, which direct details of a design. Guidelines are found in a variety of places:

- Professional, trade and academic journal articles provide a good source of information about current practice and experience.
- General handbooks offer a coherent and comprehensive coverage of the area.
- House style guides detail the standard functional and display techniques for particular computers or organizations. For example, Apple Computer's Inside Macintosh describes the use of the Macintosh style windows, scroll bars and icons.
- In a brief introduction such as this, we cannot hope to cover all the details of guidelines; however, in most reviews, a number of principles stand out.
- Know the user population. This can be difficult to achieve, especially when a diverse population of users has to be accommodated or when the user population can only be anticipated in the most general terms. Knowing the user includes being sympathetic to different user needs by, for example, providing program short-cuts for knowledgeable users, promoting the 'personal worth' of the individual user and allowing users to perform tasks in more than one way.
- Reduce cognitive load. This concerns designing so that users do not have to remember large amounts of detail. Methods for achieving this include:
    - ♦ Minimize memorization by using techniques such as selecting from a menu rather than remembering command names, using names for objects rather than numbers, and giving the user access to

(understandable) system documentation. However, care must be taken to apply principles appropriately.

The famous short-term memory limitation (that people can remember only 7 +- 2 things) does not mean that menus should be restricted to seven options! Users need not remember specific items in any detail if they can select from a displayed list.

- ♦ Minimize learning by being consistent, drawing on knowledge of similar systems and by choosing meaningful names and symbols.

- Engineer for errors. A common excuse is that a problem occurred because of "human error." But people will always make errors and indeed have to make errors in order to learn. Engineering for errors includes taking forcing actions which prevent the user from making an error (or at least make it more difficult!), providing good error messages, using reversible actions which allow users to correct their own errors and providing a large number of explicit diagnostics.
- Maintain consistency and clarity. Consistency emerges from standard operations and representations and from using appropriate metaphors that help to build and maintain a user's mental model of a system. A designer can only have ideas about what is clear based on initial information about users. Designs must be confirmed with users - through prototyping and evaluating designs — to be certain that the system's interface really is clear.

Standards

Standards concern prescribed ways of discussing, presenting or doing something. Standards seek to achieve some form of consistency across products which are of the same type. We are familiar with standards in many walks of life - standard colors for electrical wiring, standard controls on cars, standard shoe and clothing sizes. Establishing standards encourages:

- A common terminology. For example, standard measures of usability or performance mean that designers and users know that they are discussing the same concept. All systems of the same type can be subjected to a standard benchmark that facilitates comparisons.
- Maintainability and evolvability. Standard implementation techniques facilitate program maintenance because all programs can be expected to have a shared style and structure. Additional facilities can be added to a system if its external interfaces are of a standard form.
- A common identity.
- Reduction in training.

Coding Guidelines and Standards

The coding guidelines and standards used in C-ProMPT were derived from various Hewlett-Packard and Institute of Electrical and Electronics Engineers (IEEE) sources. The standards were initially modified for an academic environment. Over a period of two years, the standards were enhanced or deleted, and added based on their use in the classroom and by suggestions from students and instructors.

The C-ProMPT on-line help file describes the standards and guidelines required to create readable and maintainable source code. The following areas were incorporated within the coding standards and guidelines:

File	describes the makeup of the file header, file size, and naming conventions.
Function	describes the makeup of the function header, placement of prototypes and main (), function size, and coupling
Operators	describes the allowable dependencies on

	evaluation order, character tests, lexical rules for operators, order of side effects, and placement of parentheses
Data and Variables	describes variable naming conventions, lexical rules for variables, and constant maintainability
Control Structure	describes the lexical rules for control structure
Additional Information	describes the importance of code reviews, the glossary of terms used, design simplicity and implementation, the difference between standards and guidelines, why coding standards and guidelines are important, and side effects when writing macros

Document standards

The document guidelines and standards were derived from the Institute of Electrical and Electronics Engineers (IEEE) and the Department of Defense standards. The guidelines and standards used in C-ProMPT have gone through several years of testing with over 50 computer science and software engineering academic projects. The students provided feedback via weekly status reports (one of the standard documents) and weekly discussions and out-briefings with the students either individually or in groups. The result is the C-ProMPT Document Guidelines and Standards. These guidelines and standards fall into five document categories and two support categories. The description of each category is as follows:

Why Document Standards?	Provides background information as to why document standard usage is important for a mature practitioner or organization.

Initial Document Preparation	Describes the techniques to automate the document process and how the specific sections within the documents should be formatted.
Planning Documentation	Describes the format and topics that should be included in the Software Development Plan, Software Project Plan, and Software Test Plan and Test Procedure documents.
Software Requirements Documentation	Describes the format and topics that should be included in the Software Requirements Specification (SRS) document. Several analysis and requirement methods are discussed (functional, object-oriented, object, etc.)
Software Design Documentation	Describes the format and topics that should be included in the Software Design Documentation. Functional and object-oriented design documents are presented.
Software Test Documentation	Describes the documents required to fully satisfy the testing requirements.
Project Weekly Summary Reports	Describes the team status reports for the B.S. Computer Science and M.S. Software Engineering project courses.

Document standard templates

I designed and created several Word for Windows software engineering document templates that allow the students to quickly create and manage the changes required during a software project. These templates have gone through the same evolutionary process as the document guidelines and standards.

The document templates are accessed through the C-ProMPT *Document Topic* screen.

Software Engineering Special Topics

The *Software Engineering Special Topics* screen allows the user to select various software engineering topics that will help them to understand software analysis, design, risk analysis, unit testing, document design principles, and the Software Engineering Institute's Software and People Capability Maturity Model. The topics are broken down into three main topics, software development, software engineering references, and a catch all topic, personal and organization

Software Development

The Software Development topic provides essential information on how to write a good software requirements specification, software design fundamentals, and unit testing techniques. It contains the following:

What is a good Software Requirements Specification? — Provides the background information for writing a good SRS.

Software Design Fundamentals — Describes the various methods and notations used to adequately depict the design concepts using text and graphical notations.

Unit Testing Techniques — Describes a standard approach to software unit testing that can be used as a basis for sound software engineering practice.

Software Engineering References

The references topic contains a comprehensive glossary containing the most used Software Engineering terms and a Software Engineering bibliography partitioned into specific software engineering topics.

Personal and Organization

This area contains background information on the Personal Software Process, risk management, document design fundamentals, and formal technical reviews. It also contains a quality improvement story that provides another look at how to perform quality improvement in your project or organization.

Document Design Fundamentals — This section will teach a technical writer unfamiliar with document design some of the basic tenets of layout and typography. It will also impart to the reader a sense of what constitutes good graphic design.

Quality Improvement Story — Some people have a hard time digesting information about quality improvement or any other more technical issue unless it is put in the form of a story. The Japanese often use little parables to teach the concepts of quality improvement. The following story teaches quality improvement from just such a perspective.

Personal Software Process Background — The Personal Software Process (PSP) is a self-improvement process designed to help you control, manage, and improve the way you work. It is a structured framework of forms, guidelines, and procedures for developing software.

Risk Management — This section describes how Rockwell Collins Commercial Avionics Engineering Process and Support group implemented Risk Management into their organization. (courtesy of Art Gemmer)

SEI - CMM — Describes the Software Engineering Institute's People and Software Capability Maturity Model.

Formal Technical Review Fundamentals — This paper examines a case study of the characteristics of an organization's culture that can inhibit the effective implementation of a formal risk management program. It presents a "Learning Model" that has been used to identify and deal with organizational needs and their associated cultural issues.

Document Cost and Effort Estimation

A major part of any software project is the required engineering documentation. Estimating the effort required to produce any type of documentation requires an understanding of the problems of documentation production, publication, and maintenance.

The document and cost effort estimation used in C-ProMPT were derived from my technical writing experience and interviewing several other technical writers and editors. This estimation method is for a single technical writer, not a team of writers. However, with the addition of statistical analysis and probability formulas the estimation method could be modified for a team of writers. This estimation method is performed as follows:

First, estimate the document production schedule. As a guideline you can use the following approximate percentages of the total time spent on each document milestone:

1. Develop a document release plan that would include the scope and format. (technical writer and responsible engineer) — 2%
2. Generate draft. (technical writer and responsible engineer) — 35%
3. Review draft. (technical writer, responsible engineer, technical editor, and software manager) — 20%
4. Generate final draft. (technical writer and responsible engineer) — 25%
5. Review final draft. (technical writer, responsible engineer, technical editor, and software manager) — 3%

6. Produce end-user documentation. (technical writer and technical editor) — 15%

Once you have the schedule completed, you can then complete the document cost estimate. Various cost estimates are used in the computer industry to estimate the final cost and time required to produce a document. One such method used is as follows:

1. Estimate the page count of the document. This can be done by using the page count of similar documents or the experience of the document planner.
2. Multiply the page count by 3 to 6 hours for an update ***OR*** 5 to 8 hours for a new document. The result is the documentation completion time in hours.

   Note: The low end hours per page does not include any explicit fudge factor. The low end is usually used for experienced technical writers and it is the barebones minimum time required, barely taking time for vacations, replacements, illness, management, meetings, and so forth into account. The upper end hours per page is used for inexperienced technical writers. The upper end does account for vacations, replacements, illness, management, meetings, and so forth. Usually an experienced technical writer would use a mid-range value to estimate the time required to complete the document.
3. Divide the documentation completion time by 35 hours per week. Use weeks rather than days to calculate documentation time for large projects. Weeks and months are less complicated and more desirable for schedules.
4. Divide the documentation completion time in weeks by the number of engineering/writer/production staff available for the document. The result is the total number of weeks required to complete the document.

To determine the estimated cost breakdown for the document use the following percentage values:

- **Technical Writers** — 65% of the total hours required to complete the document times the average salary of the technical writers that will do the engineering interviews, research, writing, and review the document draft updates and modifications.
- **Engineers** — 25% of the total hours required to complete the document times the average salary of the engineers that will participate in the interviews and preliminary and final document reviews.
- **Technical Editors** — 6% of the total hours required to complete the document times the average salary of the technical editors that will check the document for correct grammar, spelling, and proper syntax.
- **Managers** — 4% of the total hours required to complete the document times the average salary of the managers that will review the preliminary and final document.
- **Overhead** — the cost required to support the above personnel. This would be equipment, work space, support personnel, and so forth.

Stand-alone C-ProMPT Help Files

The C-ProMPT help system files were designed to work as a stand-alone system. There are four help systems, PSP Process, coding guidelines and standards, document guidelines and standards, and software engineering special topics.

The C-ProMPT software application is linked to specific topics within each C-ProMPT help system. This linkage is part of the RoboHelp functionality.

Another feature of the C-ProMPT help systems is "hyperview." This feature creates a "topic tree" of the major topics contained within the help file. The user can transverse the tree by pointing and clicking on a help topic within the tree. Hyperview also allows the user to print multiple selected topics rather than one topic at a time as in the standard Windows help system.

The C-ProMPT help file system also contains an enhanced search engine that allows the user to search for any specific word and phrase. This feature is superior to the standard Windows help system, which only allows the user to search major topics or user defined key words.

Construction

Support Documentation

C-ProMPT Help Files Conversion

As I mentioned earlier, I decided to use RoboHelp to write the Windows Help files. The makers of RoboHelp designed this tool to work within Word for Windows. This was great, because it reduced the development environment complexity. There was just one problem; the guidelines and standards documentation were written in WordPerfect for Ventura Publisher. I had to strip out all the Venture Publisher embedded command verbiage and then convert the document over to Word for Windows. Essentially, I had to reformat the entire document suite. Once this was done, the Windows Help file development was straight forward.

RoboHelp is a great tool. It converts a Word document by changing the headings to help topics (you decide which headings to convert to topics) and then automatically creates the hyperlinks and topic key word links to the topics. I had all the help files created within three days. If I had created the help files manually it would have taken three to four months. It was so fun and easy to create a help file, I had time to add more software engineering development support information to C-ProMPT.

The biggest problem with the Windows Help files is the graphic format. All my graphics for the guidelines and standards suite were in GEM format. The graphic format required for the help file was WMF or Windows Meta-File format, a very simple vector graphic format. I had to convert almost a hundred GEM graphic files to WMF format. I used CorelDraw, version 3.0, as the converter. The results were disappointing. The converted documents were unusable. The time I had saved in developing the help files would now be used to redo and redraw the graphic files. I told one of my co-workers about this problem. He gave my a copy of CorelDraw, version 5.0, and said that this version should take care of the problem. This CorelDraw version did the trick. The converted GEM graphic files were acceptable.

There were a lot of minor problems to overcome while developing the help files. The Windows Help compiler can not compile Word for Windows documents; the compiler compiles another text format called Rich-Text Format or RTF. RoboHelp converts the Word for Windows documents to RTF format. The Windows Help compiler then uses the RTF files to create the Windows help files. Unfortunately, the RTF files only use the lower set of ASCII characters. That means that the “, ”, ‘, ’, and — characters plus all the symbols would not convert to RTF format. The character is stripped from the document. I had to go through all the documents and change the “ and ” characters to a " character and the ‘ and ’ characters to ' character. (so if you find a word that should be possessive and its plural you now know the reason).

Another minor problem dealt with Word for Windows. All word processors maintain a paragraph formatting table for each paragraph in the document. This formatting information includes the tab set, indentation, font size and type, bullet type, etc. All this information increases the file overhead and file size. To reduce the file size, Word doesn’t include all the paragraph formatting data for paragraphs that repeat the previous paragraphs format. This is great for Word documents but a nuisance for Word to RTF converted documents. RTF files require the formatting information for all paragraphs. So, if the Word document had a series of bullets or numbered paragraphs, the first RTF bullet or numbered paragraph was formatted correctly; the subsequent paragraphs were not. To overcome this problem, I had add an extra tab to all the odd bullet and numbered paragraphs. This little operation would trick Word and force it to add formatting information to all the document paragraphs.

It took only three days to create the Windows Help files. It took about a week to make the necessary changes to the Word documents. The Word to RTF and GEM to WMF conversion processes were under control and properly converted.

The next stage required that I add the graphic files to the help files as pop-up topics. This meant that I had to create hyperlinks in the documentation to each graphic file. Thanks to RoboHelp this was again a very straight forward procedure and did not take very long.

The first set of help files was primitive. I wanted more pizzazz. I added the main topic screen push button links to the major topics rather than a simple list of topics. The push button links required an icon and text. The text would help the user to recognize the relationship between the text and icon symbol. However, the icons had to have a visual relationship to the subject material; otherwise the users' mind would refuse to recognize the text and icon relationship. I spent a couple of weeks designing the icons and then a week revising them after getting some feedback from my co-workers. The push buttons added the pizzazz to the help file main topic screen.

The C-ProMPT help screens were completed as stand-alone help files. This was done so that the students and evaluators could test the help files before the C-ProMPT software application development began. This testing process took about a month. The students and evaluators did not find anything wrong with the help files. One evaluator did suggest that I put more information into the help file suite. The students were excited to have the guidelines and standards on-line; they discovered that they could now cut and paste the help files subject material into their documents. The C-ProMPT help files exceeded the requirements, now onto the C-ProMPT software application.

C-ProMPT Software Application

As RoboHelp reduced the effort required to develop the C-ProMPT help files, Borland's Delphi did the same with the C-ProMPT software application. Delphi is based on Object Pascal, however, the development environment is component based. In other words, you select the component (button, panel, edit box, list box, etc.) place it on the form (Windows screen) and position it where you want. Basically, that is all you need to do to program the application. Delphi writes the Pascal code and sets-up the linkage to the Windows Application Language Interface (API). Delphi hides the Windows development complexity from the programmer.

The C-ProMPT screens were constructed based on the design criteria and RoboHelp's help file linkage constraints. The "PSP Instruction," "Coding Standards," "Document Standards," and "Special Topics" screens and Windows Help file linkages were constructed within three days.

The "Document" screen took about a week to construct. This screen required the user to select either a PSP form, template, or document example; then the module would launch Word for Windows and load the document. Unfortunately, the literature on how to do this was almost non-existent, at least, for the Delphi environment. After buying four books on Delphi programming, I found a short paragraph in one reference book that referred to a non-existent feature in Delphi and said that this was easier than using the Windows API *ShellExecute* routine. This was the Windows API routine I was looking for. I still had to experiment with this routine to determine how it worked, since the routine's literature was very cryptic. I found out that I had to use the routine to launch Word for Windows and then again to get Word for Windows to load in the selected document. I now had the "Document" screen working.

There is a memory allocation problem when you run C-ProMPT and Word for Windows at the same time. C-ProMPT uses between 10 to 20% of the memory resources depending on the computer system and RAM. Word for Windows is a well-known memory hog. Word also is known to allocate memory without releasing it back to the system when Word is closed. This "feature" was confirmed when I opened and closed several instances (copies) of Word. Eventually, I would run out of memory and Word would not load. I spent several days trying to find a workaround to this problem, without success. I also constructed an elaborate exception handling routine that would tell the user why Word for Windows or the document would not load. However, when I tried using this routine Word for Windows would not load at all, due to memory allocation problems (low memory). When I removed the error handling routine, Word for Windows would load without any problem. (I'm still trying to find out why this is happening. The C-ProMPT development machine is a 486DX4 100mhz with 20 megabytes of RAM.) Although C-ProMPT does not have an internal error handling routine for launching an external application, Windows does have an error handling routine (the error messages are very terse) that will at least let the user know why Word for Windows or the Word document will not load.

The "PSP Process" screen was the last screen to construct. I intended to automate all the PSP processes and manage the data with a Borland Paradox database. Again, the literature was very cryptic on relational database usage in a Delphi application. There was a lot of information on using single tables in a form or using Structured Query Language links to external databases, but practically none on using multi-tables within one form. I find out that developing an extensive database system was not very straight forward as the Delphi hype said. If I wanted to construct the automated PSP processes, I would have to develop a relational database management system with integrity and referential checks.

I decided that this project was well beyond the scope of the PDE. I backed off the automation process and just set-up the screen as a "Software Development Phase" information screen. That is, each phase would have a push button that would indicate which forms were required for that specific development phase. I also incorporated a document production estimate algorithm that I developed back in 1991 for Hewlett-Packard. This algorithm was based on my technical writing experience and experiences of dozens of other technical writers. Since C-ProMPT is a concept prototype, I felt these changes fell in line with the initial requirements.

The total time spent developing the C-ProMPT software application took about three months. This included researching the RoboHelp and Delphi development environments and features, overcoming problems with the development environment "features," and tweaking the application. Overall, the C-ProMPT software application exceeded my expectations and the initial requirements.

Testing

Unit Tests

Unit testing was fairly straight forward. For the most part, these tests consisted of the following:

- hyper-link tests
- help file textual continuity
- screen navigation
- database linkage
- data retrieval and storage
- error handling capability

Problems:

Most of the errors found were syntactical, logic, and design errors. In all cases the defect removal took less than five minutes. While correcting some of the defects I accidentally introduced defects back into the module. This was primarily due to my misunderstanding of Windows Application Programming Interface (API) commands, Dephi programming environment, and Object Pascal.

Testing Hyperview was a very simple process; click on the Hyperview button. Unfortunately, it did not work as expected. An error message stated that the "Help

Browser" did not recognize Microsoft meta-files. A quick call to Blue Sky Software resolved the problem; the Hyperview and Hyperfind DLL files (hyprview.dll and hyprfind.dll) were defective. I downloaded the good versions via the Internet FTP services. I started the Hyperview/HyperFind test, and it ran successfully.

System Tests

The Delphi programming environment reduce the systems testing requirements. The environment keeps tabs on the modules within the project and automatically recompiles a module if it has been edited. Since this was single person prototype project working within a rapid prototyping environment, system tests were not required.

Usability Tests

C-ProMPT did require usability tests. These tests were carried out by four individuals. Refer to Section 3, Software Usability Testing, for more information.

Future Enhancements

Currently all the Personal Software Processes are manually completed by the student. I plan to fully automate the Personal Software Process. The C-ProMPT application does contain the software "phase" push buttons within the "PSP Processes" screen, but they are non-operational. The completed C-ProMPT project will automate the software "phase" processes and will perform the following functions:

1. PROxy-Based Estimating (PROBE) code size estimation tool. This code size tool is based on Watts Humphrey's description in *A Disciplined Approach to Software Engineering,* pages 117 through 141.
2. A development repository that will contain the following information and database activities to perform relational database operations in this data:

- Project time recording log. This log will contain the actual time spent in each phase of the project.
- Task plan that will allow the student to record planning estimates for each task identified to complete plus the actual time spent.
- Schedule plan that will allow the student to record the estimated start dates and time duration for each task plus the actual start dates and time duration.
- Operational scenario that describes the likely operational scenarios that will occur while the student's application is running. It is also used to specify the test scenarios.
- Function specification for each module or object.
- State specification for each module or object.
- Logic specification for each module or object.
- Log issues during project (tracking information)
- Test case specification and description for each operational scenario, function, state, and logic specification.
- Test case expected and actual results.
- Log defect discovered and removed during the project.

3. Software Usability Testing

This section describes the Software Engineering Academic Project Management Production Tools (C-ProMPT) evaluators, evaluator selection rational, evaluation forms.

C-ProMPT Evaluators

The C-ProMPT evaluators:

Cindy Powers	Computer Science undergraduate student at National University. Cindy graduated from National University in 1994. Her project team was one of three that participated in several brain storming sessions regarding the proposed C-ProMPT project. She is extremely interested in organizational and individual process improvement.
Dr. Alisher Abbullayeu	Computer Science student at National University. Alisher is a Mathematics Assistant Professor at National University with a keen interests in computers and software development. He is currently seeking a B.S. in Computer Science. Alisher attended my Personal Software Process course and is familiar with the concepts and practices incorporated with C-ProMPT.
Jeff Dunlap	Software Engineering graduate student at

	National University. Jeff graduated from National University in 1994. He was introduced to the Personal Software Process while I was teaching the course. Jeff is a project lead at Intel Corporation. He is also trying to incorporate sound software engineering practices within his group.
Dr. T. Joseph Walsh	Joe is a Management Information Systems assistant professor at Capital University. Joe has at least eighteen years experience in management information (information engineering) system development and software application development.
Harry Wheelis	A software practitioner for the State of California, Employment Development Department (EDD). Harry is a software developer at EDD. Harry understands most software engineering methods and practices. He is a member of the ad-hoc Software Engineering Process Group at EDD.

Evaluator Selection Rational

The evaluators were selected because they represent undergraduate and graduate level students, teaching faculty, and software industry. All are familiar with software engineering methods and practicies. Most have used a Computer-Aided Software Engineering or Software Process Engineering software development tool.

Human-Computer Interaction Evaluation Forms

The Human-Computer Interaction evaluation form used is strickly for usability testing purposes. In other words, the evaluators are checking the user interface for the following attributes:

HCI Attribute	Checks
Screen	Character sharpness and legibility Screen highlighting helpful Layouts are adequate Screen sequencing is predictable and easy to navigate
Terminology and system information	Terms are consistent throughout the system Terminology relationship to work environment Consistent appearance of message on screen Instructions are clear Appropriate usage of feedback messages Error message are helpful
Learning	Familiarization time required Exploration and discovery of features Memory retention of names and commands Sequencing of tasks Content and amount of help provided Tutorial and reference manual content
System capabilities	Response time System performance time Reliability of the system System physical characteristics Undo and correction capability Needs of novice and experience users
Overall User Reaction	Overall experience of the user's interaction with the system

The evaluators also describe their computer experience background, the computer system used during the test, and general comments they may have about C-ProMPT.

User-Interaction Satisfaction Results

The following information indicates the computer systems used by the evaluators, their past experience, and average rating for each usability attribute.

Type of Systems Used By the Evaluators

Type of hardware: ⓪ 386 ___ Mhz ④ 486 ___ Mhz ① Pentium ___ Mhz

Disk Operating System: ⑤ MS-DOS version ______ □ PC-DOS version ______

How long have you worked on this system?

□ less than a month	□ 1 month to less than six months
□ six months to less than year	① 1 year to less than 2 years
□ 2 years to less than 3 years	④ 3 years or more

On the average, how much time do you spend per week on this system?

□ less than one hour	□ one to less than 4 hours
② 4 to less than 10 hours	③ over 10 hours

Evaluators' Past Experience

How many different types of computer systems (e.g., Unix, VMS, Intel personal computers, Macintosh) have you worked with?

□ none	□ 1	□ 2
② 3 - 4	③ 5 - 6	□ more than 6

Of the following devices, software, and systems, check those that you have personally used and are familiar with:

⑤ mouse	⑤ text editor	⑤ word processor
⑤ workstation	④ electronic spreadsheet	⑤ interactive help system
⑤ graphics software	⓪ computer games	⑤ graphic user interface
⑤ electronic mail	② process manager software	④ project manager software
② CASE tools		

Screen

Usability Attribute	Average Rating (1 through 9)		
Characters on the computer screen	hard to read	8.8	easy to read
Image of characters	fuzzy	8.6	sharp
Character shapes (fonts)	barely legible	8.8	very legible
Was the highlighting on the screen helpful?	not at all	8.8	very much
Use of color coded hyperlinks	unhelpful	8.4	helpful
Were the screen layouts helpful?	never	8.0	always
Amount of information displayed on screen	inadequate	8.6	adequate
	adequate	2.9	too much
Arrangement of information on screen	illogical	8.4	logical
Sequence of screens	confusing	8.6	clear
Next screen in a sequence	unpredictable	8.6	predictable
Going back to the previous screen	impossible	8.8	easy
Beginning, middle and end of tasks	confusing	8.6	clearly marked

Terminology and System Information

Usability Attribute	Average Rating (1 through 9)		
Use of terms throughout system	inconsistent	8.8	consistent
Process management terms	inconsistent	8.8	consistent
Does the terminology relate well to the work you are doing?	unrelated	8.8	well related
Computer terminology is used	too frequently	8.8	appropriately
Terms on the screen	ambiguous	8.6	precise
Messages which appear on screen	inconsistent	8.6	consistent
Position of instructions on screen	inconsistent	8.2	consistent
Messages which appear on screen	confusing	8.5	clear
Instruction for commands or choices	confusing	8.2	clear
Instruction for correcting errors	confusing	7.0	clear
Does the computer keep you informed about what it is doing?	never	7.4	always
Performing an operation leads to a predictable result	never	7.6	always
User can control amount of feedback	never	6.0	always

Terminology and System Information (continued)

Usability Attribute	Average Rating (1 through 9)		
Error messages	unhelpful	6.3	helpful
Error messages clarify the problem	never	6.5	always
Phrasing of error messages	unpleasant	7.3	pleasant

Learning

Usability Attribute	Average Rating (1 through 9)		
Learning to operate the application	difficult	9.0	easy
Getting started	difficult	9.0	easy
Learning advanced features	difficult	8.8	easy
Time to learn to use the application	slow	9.0	fast
Exploration of features by trail and error	discouraging	8.8	encouraging
Exploration of features	risky	8.6	safe
Discovering new features	difficult	8.6	easy
Remembering names and use of commands	difficult	8.7	easy
Remembering specific rules about entering commands	difficult	8.3	easy

Learning (continued)

Usability Attribute	Average Rating (1 through 9)		
Can tasks be performed in a straight-forward manner?	never	8.6	always
Number of steps per task	too many	8.8	just right
Steps to complete a task, follow a logical sequence	rarely	8.4	always
Completion of sequence of steps	unclear	8.2	clear
Help messages on screen	confusing	8.8	clear
Accessing help messages	difficult	8.8	easy
Content of help messages	confusing	8.8	clear
Amount of help	inadequate	9.0	adequate
Supplemental reference materials	confusing	8.3	clear
Tutorials for beginners	confusing	8.5	clear
Reference manuals	confusing	7.8	clear
Engineering documents	confusing	8.2	clear

System Capabilities

Usability Attribute	Average Rating (1 through 9)		
Application speed	too slow	8.8	fast enough

System Capabilities (continued)

Usability Attribute	Average Rating (1 through 9)		
Response time for most operations	too slow	9.0	fast enough
The rate information is displayed	too slow	8.8	fast enough
How reliable is the system?	unreliable	8.2	reliable
Operations are	undependable	8.4	dependable
Application failures occur	frequently	8.2	seldom
Application warns the user about potential problems	never	7.7	always
Application tends to be	noisy	9.0	quiet
Computer tones, beeps, clicks, etc.	annoying	8.3	pleasant
Correcting your mistakes	difficult	8.4	easy
Correcting typos or mistakes	complex	9.0	simple
Ability to undo operations	inadequate	8.8	adequate
Are the needs of both experienced and inexperienced users taken into consideration?	never	8.4	always
Novices can accomplish tasks knowing only a few commands	with difficulty	8.6	easily
Experts can use features / shortcuts	with difficulty	8.8	easily

Overall User Reactions

Please circle the numbers which most appropriately reflect your impressions about using this computer system. Not Applicable = NA. There is room on the last page for your written comments.

Usability Attribute	Average Rating (1 through 9)		
Overall reactions to the application:	terrible	8.4	wonderful
	frustrating	8.0	satisfying
	dull	7.6	stimulating
	difficult	8.8	easy
	inadequate power	8.0	adequate power
	rigid	8.0	flexible

Evaluators' Comments

Dr. Alisher Abbullayeu

Mr. Greg Russell has accomplished a great job. Very good product.

Cindy Powers

I'd like to submit this product for testing in our application development group.

Good Work!

Jeff Dunlap

Wonderful Tool! The extensive help files were excellent. I used each Engineering Document Template to create documents for my project at work. Included is a list of problems I had with some of the templates. I loved the Coding Standards section and how you have things separated.

Harry Wheelis

Overall, the system performs its functions in a clear, concise and predictable manner. I found it very intuitive and consistent with many GUI applications I use daily in my job.

I am looking forward to evaluating the system on a desk top device where I can very screen resolution & CPU speed in order to get a better feel across . several different test scenarios. Also need to see product with database attached.

It is a product I could make immediate use of in my job, even in its current form

Dr. T Joseph Walsh

I could not access any of the forms, templates, or documents. (I could access Word 6.0a and then open them from within Word.)

It seems like the PSP Process screen promises so much and delivers so little I mean every time I saw "This is a planned PSP Process enhancement." It's well designed. It's just not complete.

You should be pleased to know the software runs under Windows 95.

4. Bibliography

The following texts, articles, and guides were considered the most important material required for this program. The material was either reviewed fully or partially during the Ph.D. program. The material was used for all aspects of the degree program including course work, course development, internship, program demonstrating excellence, personal development, and some of material I read for spin-off studies that I felt would help establish some understanding of myself and my role as a software engineer.

Index	Author & Subject Title	Description
AAA95a	*California and Nevada Tour Book*, American Automobile Association, 1995	
AAA95b	*Oregon and Washington Tour Book*, American Automobile Association, 1995	
Abbott86	Abbott, R. J. *An Integrated Approach to Software Development.* New York: John Wiley, 1986.	A general text on software engineering that is organized as a collection of annotated outlines for technical documents that are important to the development and maintenance of software.
Abdel-Hamid86	Abdel-Hamid, Tarek K., and Stuart E. Madnick. "Impact of Schedule Estimation on Software Project Behavior." IEEE Software 3, 4 (July 1986), 70-75.	The thesis of this paper is that different estimates create different projects. Schedule estimates have impact on the progress of a project, in that they directly affect staffing, training, and perceived project status. Changes in these factors due to the estimates (reduction in personnel, shifting perceptions, etc.) can backfire.
Abrahams91	Abrahams, John R., *Token Ring Networks: Design and Implementotion and Management*, NCC Blackwell, 1991	
Allworth87	Allworth, S. T., and R. N. Zobel. *Introduction to Real-Time Software Design*, 2nd Ed. New York: Springer-Verlag, 1987.	One of the few books that is devoted to this particular and rather specialized aspect of software design. The book makes good use of the concept of a virtual machine for design of such systems and is well provided with diagrams. Much of the discussion is concerned with detailed design issues. The book pulls together into a single theme material taken from diverse areas.

Index	Author & Subject Title	Description
Altman94	Altman, Ross, "Traditional Waterfall Application Development Methodology: Can It Be Shaped To Suit End-User Development?," The Working Computing Report, v17, n1 (January 1994), 9	
Anderson85	Anderson, D. R., D. J. Sweeney, and T. A. Williams. *An Introduction to Management Science*, 4th Ed. St. Paul: West Publishing Company, 1985.	An excellent text used by many management science courses dealing with the "quantitative approaches to decision making."
Archer86	Archer, Rowland, *The Practical Guide to Local Area Networks*, Osborne McGraw-Hill, 1986	
Arthur88	Arthur, Lowell Jay. *Software Evolution.* New York: John Wiley and Sons, 1988.	This book is a detailed survey of techniques for software maintenance activities. Its final chapter treats "managing for maintenance," and is good background reading for the topic when it is taught in a software project management course.
Arthur92	Arthur, Lowall Jay, *Rapid Evolutionary Development: Requirements, Prototyping & Software Creation*, New York: John Wiley, 1992	This text offers a practical, logical way to develop the next generation of business application software and improve the way a company handles information. The author provides a step-by-step guide that helps the reader to develop a system that works now but is flexible enough to grow with the needs of its users. The author uses the Plan, Do, Check, and Act processes that were first proposed by Shewart.
Arthur93	Arthur, Lowell Jay, *Improving Software Quality: An Insider's Guide to TQM*, New York: John Wiley, 1993	This text explains how to apply Total Quality Management (TQM) to software development and evolution. The author provides a good explanation of the Software Engineering Institute's Software Capability Maturity Model assessment methods and how to use this model and assessment to quickly benchmark the organization's existing software practices against the best in the world. The author then explains how to establish a baseline of excellence and implement a process improvement effort based on proven action plans. This text should be used as a supplement to Humphrey95.
Atkinson91	Atkinson, Lee and Mark Atkinson, *Using Borland C++*, Que Corporation, 1991	

Index	Author & Subject Title	Description
Author Unknown	Application Development Trends, v2, n8 (March 1994), reprint	
Azarmsa91	Azarmsa, Reza, *Educational Computing: Principles and Applications*, Educational Technology Publications, 1991	
Babbie92	Babbie, Earl, *The Practice of Social Research*, 6th edition, Belmont, CA, Wadsworth Publishing Company, 1992	This text is the most used text book for social research. Earl Babbie has included an extensive material dealing with how to review social research material. The text includes several chapters dealing with statistical analysis and how that is used in social research. This text has the same limitations as Cozby93, in that, it is limited in scope in providing a full understanding of research dealing with large systems that may have several dozens to hundred of variables
Babich86	Babich, Wayne A., *Software Configuration Management: Coordination for Team Productivity*, New York, NY. Addison-Wesley Publishing Company, 1986	This text focuses on software configuration management as a day-to-day tool for increasing programmer productivity. The author discusses the problems, solutions and principles of software configuration management.
Bach95a	Bach, James. "Let's Be Practical and Make Quality About Consequences." PCWeek, v12, n34 (August 28, 1995), A12	
Bach95b	Bach, James, "Software Quality on a Shoestring," Soft-Letter, v11, n16 (January 17, 1995), 5	
Barfield93	Barfield, Lon, *The User Interface: Concepts and Design*, Addison-Wesley, 1993	

Index	Author & Subject Title	Description
Basili87a	Basili, Victor R., Richard W. Selby, and F. Terry Baker. "Cleanroom Software Development: An Empirical Evaluation." IEEE Trans. Software Eng. SE-13, 9 (Sept. 1987), 1027-1037.	The Cleanroom software development approach is intended to produce highly reliable software by integrating formal methods for specification and design, nonexecution-based program development, and statistically based independent testing. In an empirical study, 15 three-person teams developed versions of the same software system (800-2300 source lines); ten teams applied Cleanroom, while five applied a more traditional approach. This analysis characterizes the effect of Cleanroom on the delivered product, the software development process, and the developers. The major results of this study are the following: 1) Most of the developers were able to apply the techniques of Cleanroom effectively (six of the ten Cleanroom teams delivered at least 91 percent of the required system functions). 2) The Cleanroom teams' products met system requirements more completely and had a higher percentage of successful operationally generated test cases. 3) The source code developed using Cleanroom had more comments and less dense control-flow complexity. 4) The more successful Cleanroom developers modified their use of the implementation language; they used more procedure calls and IF statements, used fewer CASE and WHILE statements, and had a lower frequency of variable reuse (average number of occurrences per variable). 5) All ten Cleanroom teams made all of their scheduled intermediate product deliveries, while only two of the five non-Cleanroom teams did. 6) Although 86 percent of the Cleanroom developers indicated that they missed the satisfaction of program execution to some extent, this had no relation to the product quality measures of implementation completeness and successful operational tests. 7) Eighty-one percent of the Cleanroom developers said that they would use the approach again. This paper can be used to illustrate quality-based life cycles.

Index	Author & Subject Title	Description
Basili87b	Basili, V. R., and H. D. Rombach. "Tailoring the Software Process to Project Goals and Environments." Proc. 9th. Intern. Conf: Software Engineering. IEEE Computer Society, 1987, 345-357.	This paper presents a methodology for improving the software process by tailoring it to the specific project goals and environment This improvement process is aimed at the global software process model as well as methods and tools supporting that model. The basic idea is to use defect profiles to help characterize the environment and evaluate the project goals and the effectiveness of methods and tools in a quantitative way. The improvement process is implemented iteratively by setting project improvement goals, characterizing those goals and the environment, in part, via defect profiles in a quantitative way, choosing methods and tools fitting those characteristics, evaluating the actual behavior of the chosen set of methods and tools, and refining the project goals based on the evaluation results. All these activities require analysis of large amounts of data and, therefore, support by an automated tool. Such a tool — TAME (Tailoring A Measurement Environment) — is currently being developed.
Beckley94	Beckley, Glen B., "TQM: Find the Red Flags Hiding in Existing Systems," Datamation, v40, n17 (September 1, 1994), 63-64	
Beizer84	Beizer, Boris, *Software System Testing and Quality Assurance,* New York, NY: Van Nostrand Reinhold Co, 1984	This is a comprehensive guide to system testing and quality assurance shows learners how to create and maintain reliable, robust, high-quality software. The author covers the gamut from unit testing to system testing, providing effective techniques for security testing, recovery testing, configuration testing, background testing, and performance testing. Integration testing strategies are also presented.
Beizer90	Beizer, Boris, *Software Testing Techniques*, 2nd edition, New York, NY: Van Nostrand Reinhold Co, 1990	This text explicitly addresses the idea that design for testability is as important as testing itself by showing the learner how to do it.
Bell85	Bell, Paula, Hightech Writing: How to Write for the Electronics Industry, New York, NY: Wiley-Interscience, 1985	This text is for technical writers, engineers, programmers, and anyone who writes about electronic and software products. The text offers good writing basics: templates that illustrate standard manual formats; step-by-step procedures for extracting, structuring, and presenting technical information; styles to match a variety of products.
Bell89	Bell, Paula and Charlotte Evans, Master Documentation, New York, NY: John Wiley & Sons, Inc., 1989	This text provides tips, techniques, and sample documents that the reader may need to design, write, and maintain effective documentation throughout the entire project life cycle.

Index	Author & Subject Title	Description
Bendifallah87	Bendifallah, S., and W. Scacchi. "Understanding Software Maintenance Work." IEEE Trans. Software Eng. SE-13, 3 (March 1987), 311-323.	Software maintenance can be successfully accomplished if the computing arrangements of the people doing the maintenance are compatible with their established patterns of work in the setting. To foster and achieve such compatibility requires an understanding of the reasons and the circumstances in which participants carry out maintenance activities. In particular, it requires an understanding of how software users and maintainers act toward the changing circumstances and unexpected events in their work situation that give rise to software system alterations. To contribute to such an understanding, we describe a comparative analysis of the work involved in maintaining and evolving text-processing systems in two academic computer science organizations. This analysis shows that how and why software systems are maintained depends on occupational and workplace contingencies, and vice versa.
Benson95	Benson, Scott E., "Software Processes: Integrating Processes for Software Excellence," Software Development, v3, n8 (August 1995), 51-54	
Berger85	Berger, J. O. *Statistical Decision Theory and Bayesian Analysis*, 2nd Ed. New York: SpringerVerlag, 1985.	Covers the foundations and concepts of statistical decision theory, including situations where data are incomplete. This book approaches statistics from the business perspective where not all of the data are available. It introduces the probability of accuracy of the data into the statistical calculations, statistics based on probable accuracy of the data (Bayesian), is not appreciated by the mathematical approach to statistics.
Berry92	Berry, Daniel M., *Academic Legitimacy of the Software Descipline*, Pittsburgh, PA: Software Engineering Institute, Carnegie-Mellon University, 1994	
Bersoff80	Bersoff, E. H., V. D. Henderson, and S. G. Siegel. *Software Configuration Management.* Englewood Cliffs, NJ, Prentice-Hall, 1980.	This book contains the most complete description of software configuration management available. It provides a fairly complete rationale for what to do and why to do it. The authors have their own conceptual breakdown of the subject that does not map one-for-one with the organization of this module. The book is also weak in clearly explaining how to do the tasks of configuration management.

Index	Author & Subject Title	Description
Bersoff84	Bersoff, E. H. "Elements of Software Configuration Management." IEEE Trans. Software Eng. 10, 1 (Jan. 1984), 79-87.	Software configuration management (SCM) is one of the disciplines of the 1980s which grew in response to the many failures of the software industry throughout the 1970s. Over the last ten years, computers have been applied to the solutions of so many complex problems that our ability to manage these applications has all too frequently failed. This has resulted in the development of a series of "new" disciplines intended to help control the software process. This paper focused on the discipline of SCM by first placing it in its proper context with respect to the rest of the software development process, as well as the goals of that process. It examined the constituent components of SCM, dwelling at some length on one of those components, configuration control.
Berzins91	Berzins, Valdis and Luqi, *Software Engineering with Abstractions*, New York, NY. Addison-Wesley Publishing Company, 1991	The authors present a systematic approach that leads the learner through the entire software development process, using formal specification language to develop large, real-time, and distributed systems in Ada This is an excellent advance text for Ada programmers and software engineers.
Berztiss88	Bertziss, Alfs T., and Mark A. Ardis. Formal *Verification of Programs. Curriculum Module SEICM-20-1.0*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Dec. 1988.	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.
Bifferstaff89a	Biggerstaff, Ted J., and Alan J. Perlis, *Software Reusability: Volume I: Concepts and Models*, New York, NY. Addison-Wesley Publishing Company, 1989	This volume provides a framework for understanding software reusability. The editors present an overview and assessment of reusability, then they present a variety of composition-based and generation-based systems that explain the principles underlying this new methodology and illustrate its critical place in large-scale programming projects.
Bifferstaff89b	Biggerstaff, Ted J., and Alan J. Perlis, *Software Reusability: Volume II: Applications and Experiences*, New York, NY. Addison-Wesley Publishing Company, 1989	The final volume in this series, provides actual case studies on reusability, as well as both quantitative and cognitive results.

Index	Author & Subject Title	Description
Birrell85	Birrell, N. D., and M. A. Ould. *A Practical Handbook for Software Development.* New York: Cambridge University Press, 1985.	Provides a good overview of the software engineering view of system development, supported by an overview of a wide range of the techniques that are available to support each phase of development. The latter half of the book covers a wide range of design issues, together with examples. The book makes particularly good use of diagrams to help make its points.
Bjorner82	Bjomer, D., and C. B. Jones. *Formal Specifications and Software Development.* Englewood Cliffs, N. J.: Prentice-Hall, 1982.	The primary concern of this text is the development of formal specifications, with emphasis being placed upon the need to be able to relate design to specification. It contains chapters by a number of authors describing aspects and applications of VDM (Vienna Development Method), presented at an advanced level and requiring some background in discrete mathematics.
Black91	Black, Uyless D., OSI: A *Model for Computer Communications Standards*, Englewood Cliffs, NJ: Prentice-Hall, 1991	
Blank83	Blank, J. and M. J. Krijger, eds. *Software Engineering: Methods and Techniques.* New York: Wiley-Interscience 1983.	A report produced by the Information Structures Subgroup of the Dutch Database Club, which aims to evaluate and compare a number of different design methods. Many of the methods will be unfamiliar to most readers, although the list does include more widely-known methods such as SADT, Warnier-Orr, and JSD. A summary of the features of each method is included. The use of an "Evaluation Matrix" as a means of presenting information about the features and application areas of a method is an interesting feature.
Blattner92	Blattner, Meera M. and Roger B. Dannenberg (eds.), *Multimedia Interface Design*, Addison-Wesley, 1992	
Block83	Block, Robert, *The Politics of Projects*, Englewood Cliffs, NJ: Yourdon Press, 1983	The author explores the political component of project and system failures. He states in the Preface, "... I realized that political interaction was not an optional activity, but rather a requirement of managers in any organization, ... that beneath the parries and thrusts of political fencing there was an underlying process, one that could be presented and taught to beginning and soon-to-be politicians." His book teaches this process, but more than teach, it makes the process come alive through clearly stated guidelines, practical examples and exercises, and vividly real-world case study.

Index	Author & Subject Title	Description
Boehm81	Boehm, B. W. Software *Engineering Economics.* Englewood Cliffs, N. J.: Prentice-Hall, 1981.	Presents an extensive motivation and treatment of software development and evolution in terms of costs, quality, and productivity issues. Among the results, Boehm indicates that personnel/team capability and other attributes of a software production setting usually have far greater affect on the quality and cost of software products than do new software engineering tools and techniques. It also presents an in-depth discussion of the development and details of the software cost estimation model, COCOMO that draws upon the extensive studies and analyses that Boehm and associates at TRW have conducted over the years.
Boehm84a	Boehm, Barry W., Terence E. Gray, and Thomas Seewaldt. "Prototyping Versus Specifying: A Multiproject Experiment." IEEE Trans. Software Eng. SE-10, 3 (May 1984), 290-302.	In this experiment seven software teams developed versions of the same small-size (2000-4000 source instruction) application software product. Four teams used the Specifying approach. Three teams used the Prototyping approach. The main results of the experiment were the following. 1) Prototyping yielded products with roughly equivalent performance, but with about 40 percent less code and 45 percent less effort. 2) The prototyped products rated somewhat lower on functionality and robustness but higher on ease of use and ease of learning. 3) Specifying produced more coherent designs and software that was easier to integrate. The paper presents the experimental data supporting these and a number of additional conclusions. This paper provides an example of an alternative development cycle.
Boehm84b	Boehm, Barry W. "Software Engineering Economics." IEEE Trans. Software Eng. SE-10, 1 (Jan. 1984), 4-21.	This paper summarizes the current state of the art and recent trends in software engineering economics. It provides an overview of economic analysis techniques and their applicability to software engineering and management. It surveys the field of software cost estimation, including the major estimation techniques available, the state of the art in algorithmic cost models, and the outstanding research issues in software cost estimation. This is a short summary of the thesis of [Boehm81].
Boehm87	Boehm, Barry W. "Improving Software Productivity." Computer 20, 9 (Sept. 1987), 43-57.	This article is an excellent short summary of the realistic factors involved in increasing the productivity of software developers. Boehm's explicit belief is that the quality of management is the most important factor in the success of a project. An extensive selected bibliography by productivity factor ("getting the best from people," "eliminating rework," etc.) is included.

Index	Author & Subject Title	Description
Boehm88	Boehm, Barry W. "A Spiral Model of Software Development and Enhancement." Computer 21, 5 (May 1988), 61-72.	Presents a new model for modeling the software process that explicitly attempts to address how to manage the risks associated with the development of different kinds of software systems. The presentation of the model focuses on addressing risk as a central component in determining how to structure the software development process is unique and worth careful examination. An excellent description of a risk-reduction life cycle that is the foundation life-cycle for many process-engineering tools.
Boger85	Boger, D. C., and N. R. Lyons. "The Organization of the Software Quality Assurance Process." Data Base (USA) 16, 2 (Winter 1985), 11-15.	This paper discusses and analyzes approaches to the problem of software quality assurance. The approaches offered in the literature usually focus on designing in quality. This can be a productive approach, but there are also benefits to be gained by establishing an independent quality assurance (QA) group to review all aspects of the software development process. This paper discusses the organization of such a group using the function of an operations auditing group as a model.
Booch87	Booch, G. R. *Software Engineering with Ada*, 2nd Ed. Menlo Park, Calif.: Benjamin/Cummings, 1987.	Describes the Ada language and its use, with particular reference to the features of Ada that support software engineering principles. Contains five examples on object-oriented design, presented in a highly readable form.
Booch91	Booch, Grady, *Object-Oriented Design with Applications*, Redwood City, CA, The Benjamin/Cummings Publishing Company, Inc., 1991	This text provides a practical guide for constructing complex object-oriented systems and provides a comprehensive description of object-oriented design methods (Booch method and notation) The methods and notations described are excellent for systems architecture but is extremely limited for class methods design. Booch's method has the same limitations as Rumbaugh91.
Brackett88	Brackett, John W. Software *Requirements. Curriculum Module SEI-CM-19-1.0*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Dec. 1988.	Capsule Description: 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: • requirements identification, • requirements analysis, • requirements representation, • requirements communication, and • development of acceptance criteria and procedures. The outcome of requirements definition is a precursor of software design.

Index	Author & Subject Title	Description
Branstad84	Branstad, Martha, and Patricia B. Powell. "Software Engineering Project Standards." IEEE Trans. Software Eng. SE-10, 1 (Jan. 1984), 73-78.	Software Engineering Project Standards (SEPS) and their importance are presented in this paper by looking at standards in generals then progressively narrowing the view to software standards, to software engineering standards, and finally to SEPS. After defining SEPS, issues associated with the selection, support, and use of SEPS are examined and trends are discussed. A brief overview of existing software engineering standards is presented as the Appendix. This paper is useful as an overview if no specific standard is used in class.
Brockschmidt94	Brockschmidt, Kraig, *Inside OLE 2,* Redmond, WA., Microsoft Press, 1994	This text provides an extremely detailed description of OLE version 2.0 and how to implement OLE features in Windows applications.
Brooks75	Brooks, Frederick. *The Mythical Man-Month: Essays on Software Engineering.* Reading, Mass.: Addison-Wesley, 1975.	This book can be regarded as being a classical presentation of the problems that may be encountered in the development and management of a large software system. As such, it should be regarded as essential preliminary reading for anyone who has little or no prior experience of programming-in the-large, or who has not been involved in project management. The book contains many important lessons for the designer, presented in a particularly readable format.
Brooks87	Brooks, Frederick P., Jr. "No Silver Bullet: Essence and Accidents of Software Engineering." Computer 24, 4 (April 1987), 10-19.	In this article Brooks discusses why software isn't improving by leaps and bounds like hardware, why it never will, and what it takes to get the most out of software development.
Browing84	Browning, Christine, Guide to Effective Software Technical Writing, Englewood Cliffs, NJ: Prentice-Hall, Inc, 1984	The author describes the important technical manuals and their functions and provides criteria for writing them. She also describes the approaches for writing a reference manual and the user manual in a step-by-step method.
Brown87	Brown, Brad. *Assurance of Software Quality. Curriculum Module SEI-CM-7-1.1,* Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., July 1987.	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.).
Budd94	Budd, Timothy A., *Classic Data Structures in C++:* Addison-Wesley Publishing Company, 1994	

Index	Author & Subject Title	Description
Budde84	Budde, R., K. Kuhlenkamp, L. Mathiassen, and H. Zullighoven, eds. *Approaches to Prototyping.* New York: Springer-Verlag, 1984.	A collection of papers from a workshop held to study the use of different forms of prototyping in systems design and development. Provides the most extensive survey of approaches to software development and evolution through the use of prototyping tools and techniques.
Budgen89	Budgen, David, *Introduction to Software Design. Curriculum Module SEI-CM-2-2.1,* Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Jan. 1989.	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.
Buhr84	Buhr, R. J. ,, Englewood Cliffs, N. J.: Prentice-Hall, 1984.	Presents and illustrates a top-down, design-oriented introduction to Ada, using a specially developed graphical design notation (the structure graph). Presentation is oriented toward concurrent programs.
Callaway95	Callaway, Erin, "Model Improvement," PC Week, v12, n24 (June 19, 1995), E7	
Calvert93	Calvert, Charlie, *Teach Yourself Windows Programming in 21 Days,* Indianapolis, Indiana, Sams Publishing, 1993	An excellent structured text to learn Windows programming using the C language within a very short time. This text provides information for learning the basics. It does not make the learner into an expert over night, that requires a lot of practice.
Cameron83	Cameron, J. R. *JSP & JSD: The Jackson Approach to Software Development,* Washington, D. C.: IEEE Computer Society Press, 1983.	A collection of articles and papers describing JSP and JSD and illustrating these methods using a range of examples of reasonable size and complexity.
Card90	Card, David N. and Robert L. Glass, *Measuring Software Design Quality,* Englewood Cliffs, NJ, Prentice-Hall, 1990	David Card presents a practical guide to software metrics. The author defines a complete metric set centered around design quality that can be extended throughout the development life cycle.
Carlini95	Carlini, James, "TQM and Reengineering Teams Need Networking Guru to Succeed," Network World, v12, n15 (April 10, 1995), 64	
Chabrow95	Chabrow, Eric R., "The Training Payoff," InformationWeek, 535 (July 10, 1995), 36-46	

Index	Author & Subject Title	Description
Champine91	Champine, George, *MIT Project Athena: A Model for Distributed Campus Computing*, Digital Press, 1991	
Charette89	Charette, Robert N., *Software Engineering: Risk Analysis and Management*, New York, NY: Intertext Publications : McGraw-Hill, 1989	
Charette90	Charette, Robert N., *Applications Strategies for Risk Analysis*, New York, NY: Intertext Publications : McGraw-Hill, 1990	
Christiansen92a	Christiansen, Donald, "Spectral Lines", IEEE Spectrum, Volume 29, Number 2 (February 1992), 19	
Christiansen92b	Christiansen, Donald, "Spectral Lines", IEEE Spectrum, Volume 29, Number 6 (June 1992), 19	
Christiansen92c	Christiansen, Donald, "Spectral Lines", IEEE Spectrum, Volume 29, Number 7 (July 1992), 25	
Coad91a	Coad, Peter and Edward Yourdon, *Object-Oriented Analysis*, 2nd Edition, Englewood Cliffs, NJ, Yourdon Press, 1991	The authors present an excellent guide to object-oriented analysis that provides detailed description of terminology and notation, how to find classes and objects, identifying structures, defining attributes, defining services, and translating the object-oriented analysis to object-oriented design notation.
Coad91b	Coad, Peter and Edward Yourdon, *Object-Oriented Design*, Englewood Cliffs. NJ,Yourdon Press. 1991	This second volume in a series of guides to object-oriented development focuses on improving design, developing the multilayer, multicomponent model, design the problem domain component, designing the human/computer interaction component, designing the task management component, and finally designing the data management component.
Coffee94	Coffee, Peter, "Automating Tasks Is the Next Productivity Gain," PC Week, v11, n45 (November 14, 1994), 52	

Index	Author & Subject Title	Description
Collofello87	Collofello, James S., and Jeffrey J. Buck. "Software Quality Assurance for Maintenance." IEEE Software 4, 9 (Sept. 1987), 46-51.	Collofello and Buck provide some insight on managing a project for prevention of software problems, as well as correction of problems through software quality assurance.
Comaford94	Comaford, Christine, "Version Control Is Not Optional, It's Requried," PC Week, v11, n44 (November 7, 1994), 24	
Comer93a	Comer, Douglas E., *Internetworking with TCP/IP, Volume I: Principles, Protocols, and Architecture*, 2nd Edition, Englwood Cliffs, NJ: Prentice-Hall, 1993	
Comer93b	Comer, Douglas E., *Internetworking with TCP/IP, Volume II: Principles, Protocols, and Architecture*, 2nd Edition, Englwood Cliffs, NJ: Prentice-Hall, 1993	
Comer94	Comer, Douglas E., *Internetworking with TCP/IP, Volume III: Principles, Protocols, and Architecture*, 2nd Edition, Englwood Cliffs, NJ: Prentice-Hall, 1994	
Connell95	Connell, John, and Linda Shafer, *Object-Oriented Rapid Prototyping*, Englewood Cliffs, NJ, Yourdon Press, 1995	This text describes the techniques, in a step-by-step tutorial format, for developing, iterating, refining, and evolving a prototype into a deliverable software application.
Connor85	Connor, D. *Information System Specification and Design Road Map*. Englewood Cliffs, N. J.: Prentice-Hall, 1985. ISBN 0-13-464868-4.	Essentially aimed at data processing style systems that are concerned with record management. Gives an overview of a number of methods based on a document library problem.
Conte86	Conte, S. D., H. E. Dunsmore, and V. Ye Shen, *Software Engineering Metrics and Models*. Menlo Park, Calif.: Benjamin/Cummings, 1986.	This is the single most complete treatment of available metric models usable in software engineering. Every project manager and instructor will need this book in his or her library.

Index	Author & Subject Title	Description
Cooksey95	Cooksey, Kathryn, "C/S Threats to QA," The Computer Conference Analysis Newsletter, n365 (June 6, 1995), 7	
Covey89	Covey, Stephen R., *The 7 Habits of Highly Effective People*, New York, NY. Simon & Schuster, 1989	The author presents a holistic, integrated, principle-centered approach for solving personal and professional problems. The author describes a step-by-step pathway for living with fairness, integrity, honesty, and human dignity, principles that will help the learner to adapt to change, and the wisdom and power to take advantage of the opportunities that change creates. This is a superb text on personal worth as well as a treatise on organizational change. The author interjects many examples of organizations that incorporate the principles presented in this text.
Covey94a	Covey, Stephen R., A. Roger Merrill, and Rebecca R. Merrill, *Principle-Centered Leadership*, New York, NY: Simon & Schuster, 1994	This book, along with Covey89, Covey93 and Humphrey95, was one of the most important and rewarding books that I read during this journey through my Ph.D. program. Not only is this book an excellent guide for self renewal it also helps to define what and how organizations can do to become learning organizations.
Covey94b	Covey, Stephen R., A. Roger Merrill, and Rebecca R. Merrill, *First Things First*, New York, NY: Simon & Schuster, 1994	This text offers a principle-centered approach that will transform the quality of everything the learner does by showing how it involves the need to live, to love, to learn, and to leave a legacy. The authors show the learners how to empower themselves to define what is truly important; to accomplish worthwhile goals; and to lead rich, rewarding, and balanced lives. This book, along with Covey89 and Humphrey95, was one of the most important and rewarding books that I read during this journey through my Ph.D. program. Not only is this book an excellent guide for self renewal it also helps to define what and how organizations can do to become learning organizations.
Cozby93	Cozby, Paul C., *Methods in Behavioral Research*, 5th edition, Mountain View, CA, Mayfield Publishing Company, 1993	This text is an introduction to behavioral research methods. It primarily looks at traditional research methods and not at research research methods used for organizational system, for example, open systems evaluation method. This text has the same limitations as Babbie92, in that, it is limited in scope in providing a full understanding of research dealing with large systems that may have several dozens to hundred of variables.
Cross84	Cross, N., ed. *Developments in Design Methodology*, New York: John Wiley, 1984.	A comprehensive summary of work in the field of design theory over the past twenty-five years. Includes important papers by J. Christopher Jones, Christopher Alexander, Herbert Simon, and Horst Rittel.

Index	Author & Subject Title	Description
Cupello88	Cupello, James M., and David J. Mishelevich. "Managing Prototype Knowledge/Expert System Projects." Comm. ACM 31, 5 (May 1988), 534-541.	Fundamental issues of technology transfer, training, problem selection, staffing, corporate politics, and more, are explored
Curtis87a	Curtis, Gail M., *Beautiful America's Oregon,* Wilsonville, OR: Beautiful America Publising Company, 1987	
Curtis87b	Curtis, B., H. Krasner, V. Shen, and N. Iscoe. "On Building Software Process Models Under the Lamppost." Proc. 9th. Intern. Conf: Software Engineering. IEEE Computer Society, April 1987, 96-103 .	Most software process models are based on the management tracking and control of a project. The popular alternatives to these models such as rapid prototyping and program transformation are built around specific technologies, many of which are still in their adolescence. Neither of these approaches describe the actual processes that occur during the development of a software system. That is, these models focus on the series of artifacts that exist at the end of phases of the process, rather than on the actual processes that are conducted to create the artifacts. We conducted a field study of large system development projects to gather empirical information about the communication and technical decision-making process that underlie the design of such systems. The findings of this study are reviewed for their implications on modeling the process of designing large software systems. The thesis of the paper is that while there are many foci for process models, the most valuable are those which capture the processes that control the most variance in software productivity and quality.
Curtis88	Curtis, Bill, Herb Krasner, and Neil Iscoe. "A Field Study of the Software Design Process for Large Systems." Comm. ACM 31 a 11 (Nov. 1 988), 1268- 1287	A rare study of how software development really takes place. The authors interviewed personnel from 17 large software projects. Using a layered behavioral models they analyze the impact on quality and productivity of shifting requirements, inadequate domain knowledge, and communication problems.
Cyert87	Cyert, R. M. Bayesian *Analysis and Uncertainty in Economic Theory.* Totowa, N.J.: Rowman & Littlefield, 1987.	Looks at statistical methods used in economics where data are incomplete
Dart87	Dart, Susan A., Robert J. Ellison, Peter H. Feiler, and Nico Habermann. "Software Development Environments." Computer20, 11 (Nov. 1987), 18-28.	This taxonomy of software development environments serves as useful reading for discussions of resource acquisition, allocation, and training.

Index	Author & Subject Title	Description
Davidson94	Davidson, Cliff I. and Susan A. Ambrose, *The New Professor's Handbook*, Bolton, MA, Anker Publishing Company, Inc., 1994	This book is an ideal resource for everyone making the transition to new faculty member in engineering and science. It provides excellent information on student learning, course planning, conducting discussions, lecturing, and developing exams and assignment.
Davis83	Davis, W. S., *Systems Analysis and Design*, Reading, Mass.: Addison-Wesley, 1983.	A presentation on analysis and design based around the use of three case studies. Each of the case studies is taken through the steps of problem definition, feasibility study, analysis, system design, and detailed design. The main emphasis of the book is on analysis rather than design, as such. The book is oriented toward business applications. The book primarily makes use of the SSA/SD approach to design.
Deitel94	Deitel, H. M. and P. J. Deitel, *C++ How to Program*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1994	
DeMarco79	DeMarco, T., *Structured Analysis and System Specification*. Englewood Cliffs, N. J.: Yourdon Press, 1979.	A readable book on structured analysis and system specification that covers data flow diagrams, data dictionaries, and process specification.
DeMarco82	DeMarco, Tom. *Controlling Software Projects*. New York: Yourdon Press, 1982.	This book concentrates heavily on metrics and the examination of quality factors as the basis for management of software projects. Essentially, DeMarco makes the case that you cannot control what you cannot measure. Useful for finding ways to integrate metrics into an organization. It also provides a realistic perspective on the psychology of software project management
DeMarco87	DeMarco, Tom and Timothy Lister. *Peopleware: Productive Projects and Teams*. New York: Dorset House, 1987.	A collection of essays on managing projects and the people who carry them out by a pair of authors with substantial project management and consulting experience. The essays challenge conventional management wisdom and emphasize how creating the right work environment can increase productivity.
DeMillo87	DeMillo, Richard A., W. Michael McCracken, R. J. Martin, and John F. Passafiume, *Software Testing and Evaluation*, The Benjamin/Cummings Publishing Company, Inc., 1987	Provides a complete survey of state-of-the-art software testing (for 1987), and is an essential reference to existing government standards and regulations on software testing.

Index	Author & Subject Title	Description
Deming86	Deming, W. Edwards. *Out of the Crisis.* Cambridge, Mass.: MIT Center for Advanced Engineering Study, 1986.	Deming advocates the transformation of American management to focus on increasing quality and productivity within organizations. He offers 14 guiding principles for achieving this transformation, and he identifies the "deadly diseases" and obstacles that currently are barriers to doing so. Deming's principles provided the framework for the development of postwar Japanese industry, and he is widely credited with guiding its tremendous success.
Desmond95	Desmond, John, "Investment in Methods, Mentoring Gives Sprint Competitive Edge," Application Development Trends, v2, n8 (August 1995), 51-56	
Devargas93	Devargas, Mario, *Local Area Networks*, 2nd Edition, NCC Blackwell, 1993	
Diagle91	Diagle, John N., *Queuing Theory for Telecommunications*, Addison-Wesley, 1991	
DoD88	DoD. *Military Standard for Defense System Software Development. DOD-STD-2167A*, U. S. Department of Defense, Washington, D.C., 29 February 1988.	Due to its completeness and maturity as the successor to military standards of the preceding two decades, this is one of the best available examples to use when discussing standards.
Downs85	Downs, T. *A Review of Some of the Reliability Issues in Software Engineering.* J. Electrical and Electronic Eng. 5, 1 (March 1985), 36-48.	This paper commences with a detailed discussion of the problems and difficulties associated with software testing. It is shown that large software systems are so complex that software companies are obliged to terminate the testing process and release such systems with every expectation that the software still contains many errors. The possibility of using statistical models as an aid to deciding on the optimum time to release software is discussed and several such models are described. The idea of "disciplined" programming as a means of reducing software error content is also described, and ancillary topics such as formal specifications and program proofs are discussed. Other concepts, such as fault-tolerant software and software complexity measures, are also briefly described. Finally, the implications of the fact that hardware is cheap and reliable and software is expensive and unreliable are discussed. It is argued that many designs currently in use defy engineering common sense.

Index	Author & Subject Title	Description
Downs88	Downs E., P. Clare, and I. Coe. *SSADM: Structured Systems Analysis and Design Method.* New York: Prentice-Hall. 1988.	SSADM is a highly prescriptive design method, with a fully defined structure and terminology. This book begins by describing the structure of the method, and then describes the activities that should be associated with each of the phases, stages, steps, and tasks involved. Written in a clear and readable style, this book makes good use of diagrams throughout.
Dreger89	Dreger, Brian J., *Function Point Analysis*, Englewood Cliffs, N. J.: Prentice-Hall, 1989	This text is a tutorial on Function Points. It provides numerous examples of various ways to count and analyze Function Points.
Drexler94	Drexler, Allan, David Sibbet, and Russell Forrester, "The Team Proformance Model." Team Building Blueprints for Productivity and Satisfaction, NTL Institute and University Associates. publication date unknown	An comprehensive article on the theories of team building and the interaction between team members. The author discusses the necessary steps needed to ensure a successful team. An excellent paper to formulate organizational and project teams.
Edgemon95	Edgemon, Jim, "Right Stuff: How to Recognize It When Selecting a Project Manager," Application Development Trends, v2, n5 (May 1995), 37-42	
Ellis86	Ellis Robert L., *Designing Data Networks*, Englewood Cliffs, NY: Prentice-Hall, 1986	
Ellis90	Ellis, Margaret A. and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Company, 1990	
Erlbaum90	Erlbaum, L., Cognition, *Education, and Multimedia: Exploring Ideas in High Technology*, 1990	
Evans83	Evans, M. W., P. H. Piazza, and J. B. Dolkas. *Principles of Productive Software Management.* New York: John Wiley, 1983.	This book describes a methodology for software management and the associated control techniques for the entire development process, as practiced by the authors at Ford Aerospace and Communications Corporation and several other companies.

Index	Author & Subject Title	Description
Fagan86	Fagan, M. E. "Advances in Software Inspections." IEEE Trans. Software Eng. SE-12, 7 (1986).	This paper presents new studies and experiences that enhance the use of the inspection process and improve its contribution to development of defect-free software on time and at lower costs. Examples of benefits are cited followed by descriptions of the process and some methods of obtaining the enhanced results. Software Inspection is a method of static testing to verify that software meets its requirements. It engages the developers and others in a formal process of investigation that usually detects more defects in the product—and at a lower cost—than does machine testing. Users of the method report very significant improvements in quality that are accompanied by lower development costs and greatly reduced maintenance efforts. Excellent results have been obtained by small and large organizations in all aspects of new development as well as in maintenance. There is some evidence that developers who participate in the inspection of their own product actually create fewer defects in future work. Because inspections formalize the development process, productivity and quality enhancing tools can be adopted more easily and rapidly.
Fairley85	Fairley, R. E. *Software Engineering Concepts.* New York: McGraw-Hill, 1985.	Describes the basic concepts and major issues of software engineering, including current tools and techniques. Contains a chapter on design that covers fundamental design concepts, including assessment criteria, design notations, and design techniques.
Fairley88	Fairley, Richard E. "A Guide to Preparing Software Project Management Plans." In Tutorial: Software Engineering Project Management, Richard H. Thayer, ed. Washington, D.C.: IEEE Computer Society Press, 1988, 257-264.	This is written in the spirit of the various IEEE standards for plans (quality assurance, configuration management, etc.). One of the best parts of this guide is the conceptual introduction to the software development process and the relation of the project plan to it.
Feller68	Feller, W. *An Introduction to Probability Theory and Its Applications*, 3rd Ed. New York: John Wiley, 1968.	This book is used to teach probability and statistics using a mathematical approach.
Ferrari78	Ferrari, D. *Computer Systems Performance Evaluation.* Englewood Cliffs, N.J.: Prentice-Hall, 1978.	This book looks at performance evaluation using measurement, simulation, and analytic techniques. It then applies these to solve problems characteristic of methodology selection, design alternatives, and product improvement.
Fisher91	Fisher, Alan S., *CASE Using Software Development Tools*, 2nd Edition, John Wiley & Sons, Inc, 1991	This text provides information on every key aspect of automated software engineering.

Index	Author & Subject Title	Description
Floyd91	Floyd, Steve, *The IBM Multimedia Handbook*, Brady, 1991	
Ford94	Ford, Gary, *A Progress Report on Undergraduate Software Engineering Education*, Pittsburgh, PA: Software Engineering Institute, Carnegie-Mellon University, 1994	
Fox82	Fox, J. M. *Software and Its Development*. Englewood Cliffs, N. J.: Prentice-Hall, 1982.	Discusses the development of large scale software.
Frank94	Frank, Robert, "Enjoying Best of Both Worlds," Software Magazine, v14, n6 (June 1994), 96-97	
Franz94	Franz, Louis A. and Jonathan C. Shih, "Estimating the Value of Inspections for Early Testing for Software Projects," Hewlett-Packard Journal, v45, n6 (December 1994), 60-67	
Freedman90	Freedman, Daniel P. and Gerald M. Weinberg, *Handbook of Walkthroughs, Inspections, and Technical Reviews*, New York, NY, Dorset House Publishing, 1990	This handbook explains exactly how to implement reviews for all sorts of product and software development. The handbook spells out procedures to conduct walkthroughs (or peer group reviews), inspections, and technical reviews, with extensive checklists for each type of material reviewed. A "must" text for the enlighten software engineer.
Freeman87	Freeman, Peter. *Software Perspectives: The System is the Message*. Reading, Mass.: Addison-Wesley, 1987.	This book is too general to be a single text for software project management courses. However, it does emphasize the importance of environments and preparing for transition of a product.
Friedman94	Friedman, Frank L. and Elliot B. Koffman, *Problem Solving, Abstraction, and Design Using C++:* Addison-Wesley Publishing Company, 1994	

Index	Author & Subject Title	Description
Gane79	Gane, C., and T. Sarson. *Structured Systems Analysis: Tools and Techniques.* Englewood Cliffs, N. J.: Prentice-Hall, 1979.	One of the more widely used books on structured systems analysis. The book discusses some of the problems in analysis, reviews graphical tools, and shows how the graphical tools fit together to make a logical model. Each tool is treated in detail, including the data flow diagram. A structured system development method that takes advantage of the tools is presented. The importance of changeability and how it may be treated is also covered.
Gause89	Gause, Donald C. and Gerald M. Weinberg, *Exploring Requirements: Quality Before Design*, New York, NY, Dorset House Publishing, 1989	The authors suggest that this text be used as a supplement to any requirements process that the organization may use, formal or informal. It is a very good supplement for Humphrey95, especially when dealing with project requirements.
Gilb88	Gilb, Tom. *Principles of Software Engineering Management.* Reading, Mass.: Addison-Wesley, 1988.	This book is designed to help software engineers and project managers to understand and solve problems involved in developing complex software systems. It provides practical guidelines and tools for managing the technical and organizational aspects of software engineering projects. This book has an excellent chapter on software risk management..
Gilb95	Gilb, Tom, "Reflections on Testing," The Computer Conference Analysis Newsletter, n365 (June 6, 1995), 7	
Glass81	Glass, Robert L., and Ronald A. Noiseux. *Software Maintenance Guidebook.* Englewood Cliffs, N.J.: Prentice-Hall. 1981.	This book is useful for background on managing maintenance.
Glass88	Glass, Robert L. *An Overview of Technical Communication for the Software Engineer. Curriculum Module SEI-CM-18-1.0*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., April 1988.	This module presents the fundamentals of technical communication that might be most useful to the software engineer. It discusses both written and oral communication. Although the information in this module is general, the bibliography contains helpful references.
Goley94	Goley, George F., IV, "Rapid Systems Development," Data Based Advisor, v12, n9 (September 1994), 44-47	

Index	Author & Subject Title	Description
Goodwin90	Goodwin, Mark, *Graphical User Interfaces in C++ & Object-Oriented Programming*, MIS: Press, 1990	
Grady87	Grady, Robert B. and Deborah L. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Englewood Cliffs, NJ, Prentice-Hall, 1994	This authors describes their company's (Hewlett-Packard) need for a measurable, controllable software process and of the professional effort the company mounted to meet that need. The authors discuss the metrics chosen, the tools used to collect and digest them, the selling job to get people involved, the metric forms, the training sequences, the documentation, and the results and costs. This is an outstanding text for anyone trying to implement a organizational process improvement effort in their organization. This text is one of my most used texts for metrics and process improvement
Grady92	Grady, Robert B., *Practical Software Metrics for Project Management and Process Improvement*, Englewood Cliffs, NJ, Prentice-Hall, 1992	The text emphasizes proven practices and results that include: those software development "rules" that are supported by measured evidence; how measurements should be tightly linked to organizational strategies; development metrics that help project managers; how metrics are used to achieve continuous process improvement; what measures are meaningful for a large organization. This is an outstanding text for software engineering practitioners, project managers, and process improvement managers. This text is one of my most used texts for metrics and process improvement.
GTWA95	*Gold Wing Touring Association Rider Education Program Director's Manual*, 1995	
Hamlet95	Hamlet, Dick, "Testing for Quality," The Computer Conference Analysis Newsletter, n365 (June 6, 1995), 7	
Handy89	Handy, Charles, *The Age of Unreason*, Boston, Mass., Harvard Business School Press, 1989	Dr. Handy shows how dramatic changes are transforming business, education, and the nature of work. This was an excellent text describing the problems and possible solutions to organizational change.
Hansen86	Hansen, K. *Data Structured Program Design*, Englewood Cliffs, N.J.: Prentice-Hall, 1986.	The main theme of this book is Orr's Data Structured Systems Development (DSSD) method, which is also compared and contrasted with the related work of Warnier and Michael Jackson (JSP). The program examples use COBOL, although a knowledge of this language is probably not essential to an understanding of the material. The book contains many examples of the use of Warnier/Orr diagrams.

Index	Author & Subject Title	Description
Harris92	Harris, Steve, *Troubleshooting Local Area Networks*, NCC Blackwell, 1992	
Harvey86	Harvey, Katherine E. *Summary of the SEI Workshop on Software Configuration Management. CMU/SEI-86-TR-5*, Software Engineering Institute, Pittsburgh, Pa., Dec. 1986.	This is good reference material for presenting basic concepts on configuration management.
Hass93	Hass, Glen and Forrest W. Parkay, *Curriculum Planning: A New Approach*, 6th edition, Boston, MA, Allyn and Bacon, 1993	This present presents the knowledge, performance competencies, and alternative strategies needed by curriculum planners and instructors at all levels of education. The text offers a variety of learning experiences for learners with wide-ranging interests, learning styles, and backgrounds. The text is divided into two parts, part one explores vital curriculum planning components: values and goals, the four bases of curriculum, and curriculum criteria, part two emphasizes application of the skills developed in part one identifies the many curriculum innovations and trends.
Hatley88	Hatley, Derek J. and Imtiaz A. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House, NY, 1988	This is an excellent casebook and practical reference for modeling the requirements and architecture of real-time and general systems. It provides guidance for the systems developer to develop large software-based systems.
Hayes87	Hayes, I, ed. *Specification Case Studies*. Englewood Cliffs, N. J.: Prentice-Hall, 1987.	A collected set of case studies that are all based upon the use of Z, providing a well-structured introduction to the use of formal methods. The section on specification of the UNIX filing system may involve sufficiently familiar material to provide a good introduction for many students.
Hekmatpour87	Hekmatpour, S. "Experience with Evolutionary Prototyping in a Large Software Project." ACM Software Engineering Notes 12, 1 (1987), 38-41.	Describes three alternative approaches to evolving the development of software systems through prototyping techniques and tools.
Hetzel88	Hetzel, Bill, *The Complete Guide to Software Testing*, Wellesley, Mass, QED Information Sciences, Inc., 1988	Dr. Hetzel discusses the concepts and principles of testing. Then he presents detailed discussions of testing techniques using examples, checklists, and case studies based on Dr. Hetzel's consulting and management experience. An excellent text..

Index	Author & Subject Title	Description
Hix93	Hix, Deborah and H. Rex Hartson, *Developing User Interfaces: Ensuring Usability Through Product & Process*, New York, NY, John Wiley & Sons, Inc., 1993	This text provides a hands-on approach to human-computer interaction design and implementation. This is a good intermediate text on human-computer interaction.
Horton94	Horton, William, *Designing and Writing Online Documentation*, 2nd Edition, New York, NY, John Wiley & Sons, Inc., 1994	This text is an excellent guide to the art and science of creating on-line documents and documentation systems. It covers human-computer interaction and extrapolates a set of universal principles that can be applied to any form of on-line documentation.
Hsieh95	Hsieh, David. "Configuration Management: Common Object Repository Environment," Data Management Review, April 1995, reprint	
Humphrey87	Humphrey, Watts S. *Managing for Innovation: Leading Technical People.* Englewood Cliffs, N.J.: Prentice-Hall 1987.	If there is a Mythical Man-Month for managers, this is probably it. Humphrey has collected his and his IBM colleagues' collective experiences in leading technical individuals and teams into this compact readable, and immediately useful book. It is best to read a chapter at a time, with some reflection between segments there is just too much in a typical chapter to absorb it adequately in conjunction with its neighbors.
Humphrey88	Humphrey, Watts S. "Characterizing the Software Process: A Maturity Framework." IEEE Software 5, 3 (March 1988), 73-79.	Humphrey presents a framework for the evolution of the software development process.
Humphrey89	Humphrey, Watts S., *Managing the Software Process. Reading,* Mass.: Addison-Wesley, 1989.	A genuine handbook for managers. It is very detailed and complete.
Humphrey95	Humphrey, Watts S., *A Discplined Approach to Software Engineering*, New York, NY. Addison-Wesley Publishing Company, 1995	Watts Humphrey scales the methods discussed in Humphrey89 down to a personal level, helping software engineers to develop the skills and habits needed to plan, track, and analyze large, complex projects. This text should be the software engineer's number text on software engineering principles and methods.
Hutchison88	Hutchison, David, *Local Area Network Architectures*, Addison-Wesley, 1988	

Index	Author & Subject Title	Description
Ibrahim95	Ibrahim, Rosalind L. and Iraj Hirmanpour, *The Subject Matter of Process Improvement: A Topic and Reference Source for Software Engineering Educators and Trainers,* Pittsburgh, PA: Software Engineering Institute, Carnegie-Mellon University, 1994	
IEEE83	IEEE. *IEEE Standard Glossary of Software Engineering Terminology.* New York: IEEE, 1983. ANSI/IEEE Std 729-1983	Provides definitions for many of the terms used in software engineering.
IEEE84	IEEE. *IEEE Standard for Software Quality Assurance Plans.* New York: IEEE, 1984. ANSI/ IEEE Std 730-1984.	
IEEE87	IEEE. *IEEE Standard for Software Project Management Plans.* New York: IEEE, 1987. IEEE Std 1058.1-1987.	
IEEE88	IEEE. *IEEE Guide to Software Configuration Management.* New York: IEEE, 1988. ANSI/IEEE Std 1042- 1987.	
Ingevaldsson86	Ingevaldsson, L. JSP: *A Practical Method of Program Design,* 2nd Ed. Bromley, Kent, U. K.: Chartwell-Bratt Ltd.. 1986.	A practical book that relates JSP concepts to a wider domain. (The reader is invited to draw structure diagrams to describe a train, a telephone directory, and other structures). This book is in a very readable style, and is well-provided with examples and exercises (and with solutions for the latter). A useful book for anyone teaching any details about JSP.
Jackson75	Jackson, M. A. *Principles of Program Design.* Orlando, Fla.: Academic Press, 1975.	Presents a semiformal approach to program design that maps the syntactic structure of a program's input into a structure for an algorithm to process that input. This can be considered as the source book for JSP, and despite the use of COBOL for the programming examples, it discusses a lot of important issues.

Index	Author & Subject Title	Description
Jackson82	Jackson, J. H., and C. P. Morgan. *Organization Theory: A Macro Perspective for Management*, 2nd Ed. Englewood Cliffs, N.J.: Prentice-Hall, 1982.	A useful book on organization theories.
Jackson83	Jackson, M. A. *System Development.* Englewood Cliffs, N. J.: Prentice-Hall, 1983.	This book contains the original description of JSD. It is built around three worked examples. Note that [Cameron83] and [Sutcliffe88] provide descriptions of a more current form of the JSD method and contain more manageable examples for students.
Jensen79	Jensen, R. W., and C. C. Tonies, eds. *Software Engineering.* Englewood Cliffs, N. J.: Prentice-Hall, 1979.	A collection of articles that are primarily oriented toward management. However, structured program design is covered.
Jones80	Jones, C. B. *Software Development: A Rigorous Approach.* Englewood Cliffs, N. J.: Prentice-Hall, 1980.	Presents a formal approach to specification and verification of programs and to the use of abstract data types. The material of this book may be difficult for anyone who lacks the necessary mathematical background or who is unfamiliar with the type of notation used.
Jones86	Jones, Capers, *Programming Productivity,* New York, NY. McGraw-Hill Book Company, 1986	Capers Jones summarizes in this text the experience of the first 30 years of commercial and industrial programming and to point out both the real progress that has occurred and the trends that the author speculated were likely to take place in the beyond 1986. This text provides a good foundation for the why and how software metrics came into being and why they are important now as when the author wrote the text.
Jones91	Jones, Capers, *Applied Software Measurement: Assuring Productivity and Quality,* New York, NY. McGraw-Hill Book Company, 1994	Capers Jones provides a complete guide to the latest methods for accurately measuring software quality, that offers a battery of scientific tools for dramatically improving scheduling, costs, and quality of software projects. Mr. Jones focuses the text on the use of Function Points for large-scale statistical analyses. This is an excellent text for organizational measurement techniques. Refer to Humphrey95 for a description of personal software measurement techniques.
Jones94a	Jones, Capers, "Some Statistics," The Computer Conference Analysis Newsletter, n337 (March 11, 1994), 9	

Index	Author & Subject Title	Description
Jones94b	Jones, Capers, *Assessment and Control of Software Risks*, Englewood Cliffs, NJ: Yourdon Press, 1994	
Jones95	Jones, Capers, "Hard Problems of Software Measurement," Application Development Trends, v2, n5 (May 1995), 25-28	
Kedzierski84	Kedzierski, B. I. "Knowledge-Based Project Management and Communication Support in a System Development Environment." Proc. 4th. Jerusalem Conf. Info. Technology., 1984, 444-451.	Describes the development of a knowledge-based approach to representing software development task chains and communications between coordinated development agents. A prototype processing support environment is described, as is its suggested use.
Kemper94	Kemper, Alfons and Guido Moerkotte, *Object-Oriented Database Management*, Englewood Cliffs, NJ, Prentice Hall, 1994	This text provides a comprehensive view of object-oriented database technology and the current research directions.
Kerin87	Kerin, R. A., and R. A. Peterson. *Strategic Marketing Problems: Cases and Comments.* 4th Ed. Boston, Mass.: Allyn & Bacon, 1987.	This text contains decision making and management case studies as they apply to marketing.
Kernighan76	Kernighan, B. W., and P. Plauger. *Software Tools.* Reading, Mass.: Addison-Wesley, 1976.	A popular guide to programming style and to the organization and design of software tools. Strongly linked to the UNIX philosophy of providing small, independent tools and linking these together to produce more powerful tools tailored for specific purposes.
Kidder81	Kidder, T. *The Soul of a New Machine.* New York: Atlantic Monthly Press, 1981.	This Pulitzer Prize-winning story describes the development life cycle of a new computing system (hardware and software) by a major computer vendor, together with the dilemmas, opportunities, and social dynamics that shaped its development. Strongly recommended as one of the few descriptions of the real organizational complexities surrounding the development of computing systems.
Klein92	Klein, Mike, *Windows Programmer's Guide to DLLs and Memory Management*, Sams Publishing, 1992	

Index	Author & Subject Title	Description
Kobara91	Kobara, Shiz, *Visual Design with OSF/Motif*, Addison-Wesley, 1991	
Kochan89	Kochan, Stephan G. and Patrick H. Wood (eds.), *UNIX Networking*, Hayden Books, 19989	
Korzeniowski95	Korzeniowski, Paul, "Household Makes a Mesured Move to Client/Server," Application Development Trends, v2, n8 (August 1995), 57-59	
Kupsh93	Kupsh, Joyce, *How to Create High-Impact Business Presentations*, NTC Business Books, 1993	
Lafore91	Lafore, Robert, *Object-Oriented Programming in Turbo C++:* Mill Valley , CA, Waite Group Press, 1991	
LaMonica95	LaMonica, Martin, "IS Looks for Process Management Tools," InfoWorld, v17, n31 (July 31, 1995), 25-26	
Lavenberg83	Lavenberg, S. S. *Computer Performance Modeling Handbook.* New York: Academic Press, 1983	This book is a collection of papers, most by Lavenberg, covering a number of modeling approaches including analysis, simulation, and validation of computer performance models. This is considered by some to be the reference manual for modeling practitioners who concentrate on hardware modeling.
Lehman85	Lehman, M. M., and L. Belady. *Program Evolution: Processes of Software Change.* New York: Academic Press, 1985.	Presents a collection of previously published papers that identify and reiterate the "laws" of large program evolution as discovered through empirical investigations at IBM and elsewhere over the preceding 10 year period. Unfortunately, many of the papers state the same data and results, and therefore limit the impact of its contribution.
Lehman87	Lehman, M. M. "Process Models, Process Programming, Programming Support." Proc. 9th. Intern. Conf. Software Engineering. IEEE Computer Society, April 1987, 14-16.	An invited paper that responds to and debates the proposal by Osterweil87 for programming the software process. His critique cites the inherent openness of software development practices and the limits of being able to characterize such practices with algorithmic languages.

Index	Author & Subject Title	Description
Linger79	Linger, R. C., H. D. Mills, and B. I. Witt. *Structured Programming: Theory and Practice.* Reading, Mass.: Addison-Wesley, 1979	Central theme is the design of mathematically correct structured programs by the use of systematic methods of program analysis and synthesis.
Lippman89	Lippman, Stanley B., *C++ Primer*, Addison-Wesley Publishing Company, 1989	
Liskov86	Liskov B., and J. Guttag. *Abstraction and Specification in Program Design.* New York: McGraw-Hill 1986.	Discusses different uses of abstractions, based largely around the programming language CLU, and with an emphasis upon the issues of programming-in-the-large. Primarily concerned with relatively detailed design issues.
Livingston92	Livingston, Brian, *Windows 3.1 Secrets*, San Mateo, CA, IDG Books Worldwide, Inc., 1992	This text describes in detail the information needed to install, optimize, and maintain Windows and Windows applications. A good text on Windows, especially the information that Microsoft conveniently forgot to include in their Windows software development kits.
Londeix87	Londeix, Bernard, *Cost Estimation for Software Development*, New York, NY. Addison-Wesley Publishing Company, 1987	This text provides a practical guide to modern cost-estimation techniques. The author describes the step-by-step estimation process guide, comparison between Putnam and Boehms approach to cost-estimation, and excellent cost-estimation exercises.
Loomis95	Loomis, Mary E. S., *Object Databases: The Essentials*, Addison-Wesley Publishing Company, 1995	This text describes how object databases fit into the spectrum of today's database technology offerings. Mary Loomis discusses the requirements that drive the development of object databases products, the kinds of applications that can best benefit from object database support, the functionality that object databases offer, and the direction the industry is taking.
Lorentzen95	Lorentzen, Bob, *The Glove Box Guide: Mendocino Coast*, Mendocino, CA: Bored Feet Publications, 1995	
Lorenz94	Lorenz, Mark and Jeff Kid, *Object-Oriented Software Metrics*, Englewood Cliffs, NJ, Prentice-Hall, 1994	This text identifies a set of meaningful metrics that will help the software engineer to develop better designs, more reusable code, and prepare better estimates. The authors chose metrics that have a high likely hood to identify anomalies as well as to measure progress.
MAA91	*Visualization in Teaching and Learning Mathematics*, Mathematical Association of America, 1991	

Index	Author & Subject Title	Description
Machiavelli13	Machiavelli, N. *The Prince.* Bantam Books, Inc., 1981. First published in 1513.	This book presents several excellent concepts related to influencing people in spite of an adverse relationship. It was written to explain how a prince should control his principality, but with only a minor change in view point, it also provides a remarkably incisive commentary on interpersonal and interorganizational relationships.
Mainwaring77	Mainwaring, William L., *Exploring the Oregon Coast,* Salem, OR: Westridge Press, 1977	
Mann88	Mann, Nancy R. "Why it Happened in Japan and Not in the U.S." Chance: New Directions for Statistics and Computing 1, 3 (Summer 1988), 8-15.	The story of how W. Edwards Deming's statistical quality-control methodology came to be embraced in Japan, after it failed to take hold in the U.S. The crucial factor, Mann asserts, was the buy-in of Japanese management. The competitive advantage this gave Japanese industry should not be lost on American software developers.
Marca88	Marca, D. A., and C. L. McGowan. *SADT: Structured Analysis and Design Technique.* New York: McGraw-Hill. 1988.	A detailed description of SADT, which makes use of a generous supply of illustrations and examples, as well as providing a number of case studies taken from different application domains. The large size format used for the book makes the examples particularly clear and readable.
Martin85	Martin J., and C. McClure. *Diagramming Techniques for Analysts and Programmers.* Englewood Cliffs, N. J.: Prentice-Hall, 1985	A useful summary of some major forms of diagrams that also provides a set of examples for a wide range of diagrammatic forms.
Martin89	Martin, James, *Information Engineering Book I Introduction,* Englewood Cliffs, NJ: Prentice-Hall, 1989	
Martin93	Martin, James, *Principles of Object-Oriented Analysis and Design,* Englewood Cliffs, NJ, Prentice Hall, 1993	James Martin provides a complete introduction to object-oriented (OO) analysis and design and how it is being used to create models for redesigning a business enterprise. Although the text provides good examples of OO analysis and design methods, it falls short in describing the transaction and transformation methods used to transverse the gap from analysis and design.
May75	May, Rollo, *The Courage to Create,* New York, NY: W.W. Norton, 1975	
McCarthy95	McCarthy, Jim, Dynamics of Software Development, Redmond, WA: Microsoft Press, 1995	The author discusses his techniques for delivering great software on time. He talks about the strategies and rules of thumb that worked for him at Microsoft Corporation as the director for the Microsoft Visual C++ Program Management Team. This is an excellent text.

Index	Author & Subject Title	Description
McLachian94	McLachian, Gordon, "The Laid Plans: Project Planning Is As Easy As 1.....2, 3, 4, 5," HP Professional, v8, n12 (December 1994), 72	
Menasce94	Menasce, Daniel A., Virgilio A. F. Almeida, and Larry W. Dowdy, *Capacity Planning and Performance Modeling*, Englewood Cliffs. N.J.: Prentice-Hall. 1994	The authors describe how capacity planning questions can be answered in a scientific manner. The authors discuss a methodology for capacity planning, intuitive solutions to simple performance models, software performance engineering, and how to calibrate and validate a performance model. This is a math extensive text, the reader should have a good understanding of statistics and quantitative analysis.
Meserve95	Meserve, Jason, "Consistency: One of Three Keys For J.P. Morgan Core Group," Application Development Trends, v2, n8 (August 1995), 60-62	
Metzger81	Metzger, Phillip W. *Managing a Programming Project*, 2nd Ed. Englewood Cliffs, N.J.: Prentice-Hall, 1981	Yet another of the ex-IBM managers that grew up with the software industry (Brooks and Humphrey are two others) puts pen to paper. Metzger has a very engaging, informal style. Part one of this book is a travelogue through the traditional phases of the waterfall life-cycle model, with instructions as to the role of the manager in each phase. Part two contains an annotated outline of the key documents produced at each step. This book serves as a good introduction to the process of software engineering in general. However, it is quite spare and should be used in conjunction with [Metzger87].
Metzger87	Metzger, Phillip W. *Managing Programming People: A Personal View.* Englewood Cliffs, N.J.: Prentice-Hall. 1987.	Metzger concentrates on the most important aspect of a software project the people involved in making the product. He devotes a chapter to each of the key types of personnel: analyst, designer, programmer, tester, support staff, and also the customer. There is a wealth of experience contained in compact spaces, and the art chosen to illustrate key points outdoes that of The Mythical Man-Month. The sections on the specific people can be matched with life cycle phases in [Metzger81].
Meyer89	Meyer, Bertrand, *Object-oriented Software Construction*, Englewood Cliffs, NJ, Prentice Hall, 1989	This book reviews both the array of techniques needed to obtain the full extent of the approach and the design of object-oriented systems, with particular emphasis on the design of effective module interfaces.

Index	Author & Subject Title	Description
Microsoft92	*Windows 3.1 Programming Tools*, Redmond, WA., Microsoft Press, 1992	This book provides detailed information and instruction for using built-in software development tools that are part of the Microsoft Windows software development kit (SDK). Topics included in the text include: creating and compiling resources, debugging applications, analyzing data, and compressing and decompressing data.
Millington81	Millington, D. Systems *Analysis and Design for Computer Applications*. New York: Halsted Press, 1981.	
Mills83	Mills, Harlan D. *Software Productivity*. Boston, Mass.: Little, Brown, 1983. Reprinted by Dorset House in 1988.	A collection of Mills' papers from the late 1960s to the early 1970s. It is possible to trace his thinking on programming team organization.
Mills86	Mills, H. D., R. C. Linger, and A. R. Hevner. *Principles of Information Systems Analysis and Design*. Orlando, Fla.: Academic Press, 1986.	This book presents a box structure approach to the design of information systems, based upon the use of "black box," "state machine," and "clear box" structures. Management issues involved in the design process are included in the presentation, although the main emphasis is on the design transformation techniques involved.
Mills88	Mills, Everald E. *Software Metrics. Curriculum Module SEI-CM-12-1.1*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Dec. 1988.	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.
Mimno95	Mimno, Pieter R., "Team Leaders: The Move to Distributed C/S Requires," Application Development Trends, v2, n5 (May 1995), 30-36	
Moad94	Moad, Jeff, "After Reengineering: Taking Care of Business," Datamation, v40, n20 (October 15, 1994), 40-43	

Index	Author & Subject Title	Description
Morell89	Morell, Larry J. *Unit Testing and Analysis. Curriculum Module SEI-CM-9-1.2*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., April, 1989.	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.
Moriarty94	Moriaty, Terry and Barbara von Halle, "Barriers and Bridges," Database Programming & Design, v7, n12 (December 1994), S43-47	
Morse86	Morse, C. A. "Software Quality Assurance." Measurement and Control 19 (1986), 99-104.	This paper introduces the subject of software quality assurance to a wider audience of engineers so they may appreciate why software quality assurance has a place of importance in the software process and therefore must be considered seriously for all software projects.
MSF93a	*Motorcycle Rider Course: Riding and Street Skills*, Motocycle Safety Foundation, 1993	
MSF93b	*Experience Rider Course*, Motocycle Safety Foundation, 1993	
Mullin89	Mullin, Mark, *Object-Oriented Program Design with Examples in C++*, New York, NY. Addison-Wesley Publishing Company, 1989	Mark Mullin provides a concise guide to the essential concepts and techniques of object-oriented design. The author clearly explains the key concepts of object-oriented programming such as objects, classes, entities, hierarchies, and inheritance. This is a good text for a novice object-oriented programmer.
Murdoch94	Murdoch, John, "Code Review: What Seperates a Good App From a Poor One," Data Based Advisor, v12, n10 (October 1994), 126-133	
Murphy94	Murphy, Kevin R. and Charles O. Davidshofer, *Psychological Testing; Principles and Applications*, 3rd edition: Englewood Cliffs, NJ: Prentice-Hall, Inc., 1994	

Index	Author & Subject Title	Description
Musa87	Musa, John D., Anthony Iannino, and Kazuhira Okumoto, *Software Reliability: Measurement, Prediction, Application*, New York, NY. McGraw-Hill Book Company, 1987	There are four parts to this text. Part 1 provides an introductory overview of software reliability measurement. The second part proceeds to give a practical guide for apply software reliability measurement in such areas as: system engineering, software project monitoring, scheduling and planning, software change management, and evaluation of software engineering technology. Part three discusses the underlying theoretical principles and the last part provides an evaluation of the state of the art and suggestions for further research.
Myers78	Myers, G. J. *Composite Structure Design*. New York: Van Nostrand, 1978.	A data flow approach to program design similar to Yourdon79
Myers79	Myers. G. J. *The Art of Software Testing*. John Wiley & Sons, 1979.	This is a landmark book on the principles of software testing. The self-assessment given in the foreword of the book provides real enlightenment regarding the difficulty of developing comprehensive test cases.
Nagler93	Nagler, Eric, *Learning C++: A Hands-On Approach*, West Publishing Company, 1993	
Nevin94	Nevin, Howard, "The Dynamics of Change Aren't Always Obvious," Government Computer News, v13, n17 (August	
Norton92	Norton, Daniel A., *Writing Windows Device Drivers*, New York, NY. Addison-Wesley Publishing Company, 1992	This book explains device drivers and how to write them for the Windows environment. It examines the differences between DOS and Windows drivers, then details the different Windows operating modes and the three types of Windows device drivers, system, printer, and virtual.
Olson93	Olson, Dave, *Exploiting Chaos: Cashing in on the Realities of Software Development*, New York, NY: Van Nostrand Reinhold, 1993	This is a good guide for programmers and programming managers who want to break free from uncreative software development procedures. The text uncovers the areas of order within disorder in programming, and explains how to use them to make software more productive, reliable, and responsive to customer needs.
Osterweil87	Osterweil, L. "Software Processes are Software Too." Proc. 9th. Intern. Conf. Software Engineering. IEEE Computer Society, April 1987, 2- 13 .	Describes an innovative approach to developing operational programs that characterize how software development activities should occur and how tools can be used to support these activities.
Ould90	Ould, Martyn A., *Strategies for Software Engineering: The Management of Risk and Quality*, John Wiley & Sons, 1990	

Index	Author & Subject Title	Description
Page-Jones80	Page-Jones, M. *The Practical Guide to Structured Systems Design.* Englewood Cliffs, N. J.: Yourdon Press, 1980.	Presents the tools of structured analysis and shows how to use these tools. Defines the activity of design and the qualities of a good design with respect to partitioning, coupling, and cohesion. Presents a discussion on transform and transaction analysis.
Pajerski95	Pajerski, Rose, "Software Process Improvement," The Computer Conference Analysis Newsletter, n365 (June 6, 1995), 11	
Perlman88	Perlman, Gary. *User Interface Development. Curriculum Module SEI-CM-17-1.0,* Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., April 1988	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.
Peters81	Peters, L. J. *Software Design: Methods and Techniques.* Englewood Cliffs, N. J.: Yourdon Press, 1981.	The first two chapters of this book give a very good description of the software design process, viewed as a problem-solving process. The issues of design representation are also discussed in some detail. The later chapters on design methods are now a little dated, in terms of the selection of methods used.
Peterson87a	Peterson, G. E, ed. *Object-Oriented Computing, Volume 1: Concepts.* Washington, D. C.: IEEE Computer Society Press, 1987.	A useful collection of papers concerned with the development of object-oriented thinking. It also manages to strike a balance between the view of Smalltalk-80 and that of languages such as Ada.
Peterson87b	Peterson, G. E, ed. *Object-Oriented Computing, Volume 2: Implementations.* Washington, D. C.: IEEE Computer Society Press, 1987.	Complements the material of Volume 1 by assembling papers concerned with making use of object-oriented thinking in various forms of systems.
Peterson92	Peterson, Mark, *Borland C++ Developer's Bible,* Mill Valley , CA: Waite Group Press, 1992	
Petzold92	Petzold, Charles, *Programming Windows 3.1,* 3rd edition, Redmond, WA., Microsoft Press, 1992	A basic text that describes the Windows API and implementation issues. It also describes the DDE management library, TrueType fonts, some OLE features.

Index	Author & Subject Title	Description
Pietrek93	Pietrek, Matt, *Windows Internals: The Implementation of the Windows Operating Environment*, New York, NY. Addison-Wesley Publishing Company, 1993	The author describes the internal complexity and power of Windows in a clear and concise style. He uses pseudocode to show in detail what happens when a Windows program executes. The topics include a walk through a typical Windows application, memory management, the creation and destruction of a program dynamic linking, the Windows-DOS interface, the scheduler, the messaging system, resource management, and GDI basics.
Pohl89	Pohl, Ira, *C++ for C Programmers*, Redwood City, CA: Benjamin/Cummings Publishing Co., 1989	
Pollack82	Pollack, S. V., "The Development of Computer Science," 1-51. *Studies in Computer Science*, Volume 22 of *Studies in Mathematics*, Washington, D.C: The Mathematical Association of Ameɩica, 1982,	
Pooch91	Pooch, Udo W., *Telecommunications and Networking*, CRC Press, 1991	
Prata91	Prata, Stephen, *The Waite Group's C++ Primer Plus*, Mill Valley , CA: Waite Group Press, 1991	
Preece93	Preece, Jenny, ed., *A Guide to Usability: Human Factors in Computing*, New York, NY. Addison-Wesley Publishing Company, 1993	This text presents a clear, concise account of human factors in computing and provides an excellent balance between the technical and psychological issues in human-computer interaction. This text is an excellent supplement to Shneiderman92 and Hix93
Pressman88	Pressman, Roger S. *Making Software Engineering Happen: A Guide for Instituting the Technology*. Englewood Cliffs, N.J.: Prentice-Hall, 1988.	The author of a popular software engineering textbook here addresses the problem of how to introduce software engineering techniques and tools into the workplace. In this guidebook for managers, Pressman introduces and discusses the "software engineering implementation life cycle."

Index	Author & Subject Title	Description
Pressman92	Pressman, Roger S., *Software Engineering: A Practitioner's Approach*, 3rd edition, New York: McGraw-Hill, 1992	The text is structured in five parts. Part one presents a thorough treatment of software project management issues. Part two describes analysis fundamentals and requirements modeling methods and notation. Part three presents both conventional and object-oriented design methods. Part four stresses the activities that are applied to ensure quality throughout the software engineering process. Part five discusses the impact of CASE on the software development process. A similar text is Sommerville92.
Putnam92	Putnam, Lawrence H., and Ware Myers, *Measures for Excellence: Reliable Software on Time, Within Budget*, Englewood Cliffs, Yourdon Press, 1992	This book provided quantitative software management methods and advice essential for building healthy software projects. The authors discussed life-cycle models, cost estimating, life-cycle management, productivity analysis, tracking and control.
Quarterman90	Quarterman, John S., *The Matrix: Computer Networks and Conferencing Systems Worldwide*, Digital Press, 1990	
Radice88	Radice, R. A., and R. W. Philips. *Software Engineering*. Englewood Cliffs, N.J.: Prentice-Hall, 1988.	An industrial approach to software engineering.
Ray93	Ray, Michael and Alan Rinzler, *The New Paradigm in Business: Emerging Strategies for Leadership and Organizational Change*, New York, NY: G.P. Putnam's Sons, 1993	
Reiss88	Reiss, Lcvi and Joseph Radin, *X Windows Inside and Out*, McGraw-Hill, 1988	
Riehle94	Riehle, Richard, "The Road Not Taken", HP Professional, v8, n4 (April 1994), 56-57	
Robertson93	Robertson, Lesley Anne, *Simple Program Design*, 2nd edition, New York, NY, Boyd & Fraser Publishing Company, 1993	This is an excellent text for beginner software engineers and programmers who want to develop good programming skills for solving common business problems.

Index	Author & Subject Title	Description
Rout92	Rout, T.P., "The Culture of Quality and Software Engnineering Education", SIG Computer Science Education, ACM Press, Volume 24, Number 2 (June 1992), 29, 31, and 34	
Rumbaugh91	Rumbaugh, James and et.al., *Object-Oriented Modeling and Design*, Englewood Cliffs. NJ,Prentice-Hall, 1991	This text emphasizes that object-oriented technology is more than just a way of programming. It applies techniques to the entire software development cycle. It presents a new object-oriented software development methodology — from analysis, through design, to implementation. This is an excellent text on object-oriented analysis and design notations and methods. As of now the translation between analysis and design is very weak. The author is working with Grady Booch to rectify this weakness.
Sanders95	Sanders, Lawrence G., *Data Modeling*, New York, NY, Boyd & Fraser Publishing Company, 1995	The author describes how data modeling can be used to design large and small organizational databases. This text is an excellent first time book for novice data modelers.
Santifaller91	Santifaller, Michael, *TCP/IP and NFS: Internetworking in an UNIX Environment*, Addison-Wesley, 1991	
Sarna93	Sarna, David E. Y. and George J. Febish, *PC Magazine Windows Rapid Application Development*, ZD Press: 1993	
Sarna94	Sarna, David E. and George J. Febish, "What Makes a GUI Work?," Datamation, v40, n14 (July 15, 1994), 29-30	
Sauer81	Sauer, C. H. and Mani K. Chandy. *Computer Systems Performance Modeling.* Englewood Cliffs, N.J.: Prentice-Hall, 1981.	This book is an interesting treatise on performance modeling. It covers general principles, Markovian and other queuing models, approximation techniques, simulation, measurement, and parameter estimation. It contains six modeling case studies and discusses the management aspects of modeling projects.

Index	Author & Subject Title	Description
Scacchi87	Scacchi, Walt. *Models of Software Evolution: Life Cycle and Process. Curriculum Module SEI-CM-10-1.0*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Oct. 1987.	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 lifecycle 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.
Schach93	Schach, Stephen R., *Software Engineering*, 2nd Edition, Richard D. Irwin, Inc., 1993	
Seaman89	Seaman, Don F. and Robert A. Fellenz, *Effective Strategies for Teaching Adults*, Merril Publishing Company, 1989	
Sebasta93	Sebasta, Robert W., *Concepts of Programming Languages*, 2nd Edition, Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1993	This test provides a comprehensive, up-to-date presentation of the principles, paradigms, designs, and implementations of modern programming languages. This conceptual perspective prepares the reader to critically evaluate existing and future languages and their constructs.
SEI-CCM-WS95	*Software Engineering Institute Capability Maturity Model*, version 2.0, Workshop, February 1995	
SEI-RC95	*Softwre Engineering Institute Risk Conference Proceedings*, Software Engineering Institute, November 1995	
SEI-SPIN95	*SEI SPIN Directory Fall 1995*, Software Engineering Institute, 1995	
Senge90	Senge, Peter M., *The Fifth Discipline*, New York, NY, Doubleday, 1990	Dr. Senge describes the concepts and principals of a learning organization. This text identifies organizational learning disabilities and possible solutions to overcome them. This text provides the foundation principles and practices for any organization trying to change to a learning organization incorporating total quality management principles.

Index	Author & Subject Title	Description
Senge94	Senge, Peter M., Richard Ross, Bryan Smith, Charlotte Roberts, Art Kleiner, *'i he Fifth Discipline Fieldbook*, New York, NY, Doubleday, 1994	This text is a continuation of Dr. Senge's The Fifth Discipline. This fieldbook is an pragramtic guide. It shows how to create an organization of learners by providing exercises, examples, and checklists. This is a must text for anyone dealing with organizational restructuring and total quality management concepts.
Shannon75	Shannon, R. E. *Systems Simulation: The Art and Science.* Englewood Cliffs, N.J.: Prentice-Hall, 1 975.	This book covers the life cycle of system simulation. It goes from systems investigation through validation and analysis. It covers management aspects, model translation, planning and design of experiments. It includes six case studies in simulation.
Shere88	Shere, K. D. *Software Engineering and Management.* Englewood Cliffs, N.J.: Prentice-Hall, 1988.	This book is intended for the computer professional who needs to gain a system-level perspective of software development. It contains seven chapters on the system development life cycle, including discussions of risk management and cost estimation. It uses a case study to discuss structured design and database design and then addresses such subjects as quality assurance, capacity planning, and reliability. It concludes with a "case study of a systems engineer and integration job."
Shiller90	Shiller, Larry, *Software Excellence*, Englewood Cliffs. NJ,Yourdon Press. 1990	This book is designed to specifically address and develop the notion of software excellence and how to achieve it. The author divided the book into three parts. Part one provides a solid foundation in the guiding principles of software development. Part two specifies procedures that a developer can use right away to achieve software excellence. Part three describes a set of tools used in the part two.
Shlaer88	Shlaer, Sally, and Stephen J. Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data*, Englewood Cliffs, NJ, Yourdon Press, 1988	This book lays the groundwork for an object-oriented approach to systems development through information modeling. The approach focuses on identification, formalization, and verification of expert knowledge from diverse business, engineering, and technical disciplines as a means of determining the intrinsic information requirements of the system.
Shneiderman92	Shneiderman, Ben, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, New York, NY. Addison-Wesley Publishing Company, 1992	This text provides the most complete and the most current introduction to user interface design. The author discusses the underlying issues, principles, and empirical results, and describes practical guidelines and techniques necessary to realize an effective design. This is an advance text on human-computer interaction.
Smith90	Smith, Connie U., *Performance Engineering of Software Systems*, New York, NY. Addison-Wesley Publishing Company, 1990	This text examines the performance implications of software requirements and design alternatives and provides a solid basis for assessing performance before coding begins.

Index	Author & Subject Title	Description
Smith94	Smith, Dex, "Developing Online Application Help," Hewlett-Packard Journal, v45, n2 (April 1994), 90-95	
Sommerville92	Sommerville, Ian, *Software Engineering,* 4th edition, New York, NY. Addison-Wesley Publishing Company, 1992	An excellent text that introduces the learner to a spectrum of state-of-the-art software engineering techniques which can be applied to practical software projects. A similar text is Pressman92.
Stallings88	Stallings, William, *Data and Computer Communications*, 2nd Edition, MacMillan, 1988	
Stevens91	Stevens, Wayne, *Software Design: Concepts and Methods*, Englewood Cliffs. N.J.: Prentice-Hall. 1991	The author presents an introduction to the most important software design methods available. He discusses the key software design methods' notation and issues of software design.
Strauss94	Strauss, Susan H. and Robert G. Ebenau, *Software Inspection Process*, New York, NY. McGraw-Hill Book Company, 1994	The authors provide a guide that allows software engineers to catch and resolve problems early in the design and development phases. They describe a step-by-step overview of the inspection process by showing how to integrate inspections into existing development procedures, defining inspection parameters, manage the inspection process across the scope of an entire project, select appropriate inspection data and train personnel in its use, and fine-tune the inspection process for software, hardware, and documentation development projects. This is another key text. This text alone, if successfully implemented, can increase productivity and reduce costs.
Sutcliffe88	Sutcliffe, A. *Jackson System Development.* New York: Prentice-Hall, 1988..	A clear introduction to the concepts and use of JSD. A particularly useful feature is the inclusion of two worked examples at the back of the book.
Symons88	Symons, Charles R. "Function Point Analysis: Difficulties and Improvements." IEEE Trans. Software Eng. SE-14, 1 (Jan. 1988), 2-11.	The method of Function Point Analysis was developed by Allan Albrect to help measure the size of a computerized business information system. Such sizes are needed as a component of the measurement of productivity in system development and maintenance activities, and as a component of estimating the effort needed for such activities. Close examination of the method shows certain weaknesses, and the author proposes a partial alternative. The paper describes the principles of this "Mark II" approach, the results of some measurements of actual systems to calibrate the Mark II approach, and conclusions on the validity and applicability of function point analysis generally. This article is excellent for the presentation and contrast of the two function point methods (Albrecht's and Symons's)

Index	Author & Subject Title	Description
Tanenbaum88	Tanenbaum, Andrew S., *Computer Networks*, 2nd Edition, Englewood Cliffs, NJ: Prentice-Hall, 1988	
Thayer88	Thayer, R. H., ed. *Tutorial: Software Engineering Project Management.* Washington, D.C.: IEEE Computer Society Press, 1988.	This tutorial contains many of the important papers relevant to software engineering and project management. Included are papers on software engineering, project management, planning, organizing, staffing, directing, and controlling a software engineering project.
Tomayko87	Tomayko, James E. *Software Configuration Management. Curriculum Module SEI-CM-4-1.3*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 1987.	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.
Tully84	Tully, C. "Software Development Models." Proc. Software Process Workshop. IEEE Computer Society, 1984, 37-44.	This paper discusses information systems, and the system development process, and presents a number of models both of systems and of system development. It also presents one of the few descriptions of the incremental release model of software development practiced by many large system development organizations.
Walpole85	Walpole, R. E., and R. H. Myers. *Probability and Statistics for Engineers and Scientists*, 3rd Ed. New York: MacMillan, 1985.	The mathematical approach to probability and statistics.
Walsh94	Walsh, T. Joseph, *Operations Management Decision Support System*, Project Demonstrating Excellence, The Union Institute, 1994	This is the contextual piece of Joseph Walsh's Ph.D. dissertation (Project Demonstrating Excellence). The contextual piece is an excellent of a technology PDE, which are few and far between at The Union Institute.
Warfield85	Warfield, Ron, *A Guide to Crater Lake: The Mountian That Used To Be*, Crater Lake Natural History Association, 1995	
Warnier80	Warnier, J. D. *Logical Construction of Programs.* New York: Van Nostrand, 1980..	Presents a semiformal approach to program design that maps the structure of a program's input into a structure for an algorithm to process the input.

Index	Author & Subject Title	Description
Weinberg71	Weinberg, G. M. *The Psychology of Computer Programming.* New York: Van Nostrand, 1971	This book is a classic that looks at the human element of computer programming. It investigates in detail the behavior and thought processes of computer programmers at the time. This book is still, in part, relevant for today's computer programmers. It provides a good foundation for understanding the computer programmer.
Weinberg86	Weinberg, Gerald M. *Becoming a Technical Leader.* New York: Dorset House, 1988.	Weinberg presents his "MOI" model of technical leadership in this "how-to" book. The successful problem-solving leader, he asserts, has strong skills in three areas: motivation, organization, and innovation.
Weisbord88	Weisbord, Marvin R. *Productive Workplaces: Organizing and Managing for Dignity, Meaning, and Community.* San Francisco: Jossey-Bass, 1988.	Weisbord reviews the significant movements in management science and offers his own view of how to design and manage more productive workplaces that meet more successfully the needs of both organizations and employees. The particular significance of this approach for software managers is that it recognizes the rapid changes that occur in the modern workplace and incorporates this reality into its management guidelines.
Weiss94	Weiss, Mark Allen, *Data Structures and Algorithm Analysis in C++*, Redwood City, CA: Benjamin/Cummings Publishing Company, 1994	
Whitten95	Whitten, Neal, *Managing Software Development Projects*, 2nd edition. New York: John Wiley, 1995	This book collects the experience and wisdom of virtually thousands of people and hundreds of projects and attempts to present this treasure of information in a format that allows the learner to learn from the misfortunes and successes of others.
Wiegers95	Wiegers, Karl, "Improving Quality with Software Inspections," Software Development, v3, n4 (April 1995), 55-63	
Wiener84	Wiener, R. S., and R. F. Sincovec. *Software Engineering with Modula-2 and Ada.* New York: John Wiley, 1984.	Examines each phase of the software engineering process. The focus is on object-oriented design, with implementation in Modula-2 or Ada. Presents a review of design methods and principles.
Wiener88	Wiener, Richard S. and Lewis J. Pinson, *An Introduction to Object-Oriented Programming and C++*, Addison-Wesley Publishing Company, 1988	

Index	Author & Subject Title	Description
Wiener90	Wiener, Richard S. and Lewis J. Pinson, *The C++ Workbook*, Addison-Wesley Publishing Company, 1990	
Wilkes95	Wilkes, Maurice V., Computing Perspectives, San Francisco, CA: Morgan Kaufmann Publishers, Inc., 1995	
Wirfs-Brock90	Wirfs-Brock, Rebecca, Brian Wilkerson, and Lauren Wiener, *Designing Object-Oriented Software*, Englewood Cliffs, NJ, Prentice Hall, 1990	The authors describe the basic principles for object-oriented software design by providing a coherent model for the design process, tools, examples, and exercises. This is a well cited text.
Wozniewicz95	Wozniewicz, Andrew J. and Namir Shammas, *Teach Yourself Dephi in 21 Days*, Indianapolis, Indiana, Sams Publishing, 1995	An excellent structured text to learn Windows programming using the Delphi language within a very short time. This text provides information for learning the basics. It does not make the learner into an expert over night, that requires a lot of practice.
Yourdon79	Yourdon, E., and L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and System Design.* Englewood Cliffs, N. J.: Prentice-Hall, 1979.	Presents a data flow approach to program design similar to [Myers79]. Much of this material is an expansion of the ideas expressed in [Stevens74].
Yourdon85	Yourdon, E. *Structured Walkthroughs*, 3rd Ed. New York: Yourdon Press. 1985.	A very readable book that discusses a particular way of managing the process of design and assessing the product. Reviews can be used with all methods, and this book offers some practical advice about how to organize them.
Yourdon93	Yourdon, Edward, *Decline & Fall of the American Programmer*, Englewood Cliffs. NJ,Yourdon Press. 1993	In this text, Edward Yourdon demonstrates how U.S. software organizations can become world-class shops if they exploit the key software technologies of the 1990s. The author discusses how these companies can increase their productivity and quality if they companies master these new technologies. This text is somewhat similar to Humphrey89 and Humphrey95. It is another "must" text for the enlighten software engineer.
YourdonInc93	*Yourdon Systems Method: Model-Driven System Development*, Yourdon Inc., Englewood Cliffs. NJ,Yourdon Press. 1993	This text provides the practical means by which systems can be effectively developed and maintained. It describes the YOURDON approach to software engineering. This text is for advanced software engineers.

Index	Author & Subject Title	Description
Zawrotny95	Zawrotny, Stan, "Demystifying The Black Art of Project Estimating," Application Development Trends, v2, n7 (July 1995), 36-44	