Software Engineering and Management

Kenneth D. Shere

PRENTICE HALL ADVANCED REFERENCE SERIES

SOFTWARE ENGINEERING AND MANAGEMENT

KENNETH D. SHERE
Avtec Systems, Inc.

PRENTICE HALL, Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging-in-Publication Data

SHERE, KENNETH D., (date)
Software engineering and management.

Bibliography.
Includes index.
1. Computer software—Development. 2. Software maintenance. I. Title.
QA76.76.D47S49 1988 005 87-3578
ISBN 0-13-822081-6

Editorial/production supervision
and interior design: *Joan McCulley*
Manufacturing buyer: *Gordon Osbourne and Paula Benevento*
Cover design: *Karen Stephens*

The Publisher offers discounts on this book when ordered in bulk quantities. For more information write:

Special Sales/College Marketing
Prentice-Hall
College Technical and Reference Division
Englewood Cliffs, New Jersey 07632

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 0-13-822081-6 025

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S. A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Prentice Hall Advanced Reference Series

Computer Science

DUBES AND JAIN *Algorithms for Clustering Data*
SHERE *Software Engineering and Management*

Engineering

FERRY, AKERS, AND GREENEICH *Ultra Large Scale Integrated Microelectronics*
JOHNSON *Lectures on Adaptive Parameter Estimation*
MILUTINOVIC *Microprocessor System Design GaAs Technology*
WALRAND *Introduction to Queueing Networks*

Science

BINKLEY *The Pineal: Endocrine and Nonendocrine Function*
CAROZZI *Carbonate Depositional Systems*
EISEN *Mathematical Methods for Biology, Bioengineering, and Medicine*
FRASER *Event Stratigraphy*
WARREN *Evaporite Sedimentology*

Preface

This book is intended for the computer professional who needs to gain a system-level perspective of software development. This person may have a B.S. degree in computer science, work experience in a special area (such as data bases or operating systems), or responsibility for managing software-related products. The approach taken is very pragmatic.

There are a few unique features of this book. These include a new concept for a software development and maintenance environment and a new method for applying the concepts of expert systems to conventional data bases. Most importantly, this book offers guidance not generally included as part of on-the-job training or in university curricula.

After reading and studying this book, technical staff will have a better understanding of how their technical tasks fit into the larger scheme of software development and maintenance. Managers and task leaders will be better able to control their software projects. Insights into technical and management risk, cost estimation, the utilization of a system's legacy, and other aspects of the development process are provided.

The first seven chapters are intended to provide an understanding of the system development life cycle. After studying these seven chapters, you should be able to think at the system level. On the surface, this seems simple, but it isn't. If you have been working in the details of one (or a few) specific areas, it is very difficult to suddenly be able to step back and view the entire system. Planning the activities needed to develop or integrate an entire system is not an obvious process.

Recent college graduates may know a great deal about languages and may have built compilers in a class, but performing a requirements analysis, estimating costs, and determining risks are beyond the information presented in undergraduate curricula. A substantial amount of time is wasted because even experienced professionals frequently do not

know what they are expected to produce. When you understand the life cycle, you also have a reasonable understanding of what products need to be produced and when they need to be produced. This understanding helps to eliminate floundering time at the beginning of many tasks

Chapter 8 is a case study. It serves as an introduction to the structured design techniques and data-base design, the subjects of Chapters 9 and 10, respectively. Because of the system-level orientation of this book, techniques related to quality assurance (which includes configuration management and testing), capacity planning, and reliability are discussed in Chapters 11 and 12. Having devoted most of this book to processes associated with software development, a case study of a systems engineering and integration job is presented in Chapter 13.

This sudden change in orientation emphasizes that:

- The project management procedures discussed in this book apply to all systems jobs.
- Taking existing software into account is becoming standard operating procedures; in the years to come, this case study will become the typical way of building systems.
- Once you have begun thinking at the system level, you really are a systems (or software) engineer.

Most of the chapters (and many of the sections) of this book could easily be expanded into an entire book. Consequently, it has been necessary to blend an overview of the topic with some depth. I have tried to make this blend appropriate by using practical examples of how the techniques are applied.

This book is suitable for use as a reference in a training course on software engineering. A syllabus for a very fast moving three-day training course could cover most of the material in Chapters 1, 3 to 9, 13, and 14. The target audience for this type of training course is directors of software, project managers, task leaders, and software users and developers.

This book is also suitable for use as a textbook for graduate courses on either software project management or software engineering. The manuscript for this book was used to teach software engineering to graduate students at George Mason University. We generally covered one chapter per week. It was usually impossible to lecture on all the information in a chapter in one week. This approach required the students to read each chapter in advance of the lecture. The lecture then covered the highlights of the chapter and answered students' questions. This approach is suitable for a class of good engineers who are generally at the level of a Ph.D. candidate. Most of the students worked full time and had their own experience on which to draw.

In the case of a software project management course, the pace would have to be slower. Significant portions of the manuscript for this book were used in a course with that title at the University of Maryland. The recommended chapters would be the same as the chapters covered in the training course described previously, plus Chapter 2. The pace would be slower than the pace for the software engineering course, and the instructor should cover the chapters more thoroughly. In this case, the instructor needs to emphasize

the management aspects of these chapters. I would recommend that the instructor of a project management course digress from this book occasionally to discuss problems involved with managing people.

Exercises are scattered throughout this book. They are designed to evoke thought. The reader should work the exercises as he or she comes to them. Sometimes the exercises do not have a unique answer. By skipping the exercises and reading my answer, the reader may deprive himself or herself of the chance for independent thinking. Some of the exercises may take an excessive amount of time to answer completely. I suggest that time limits be imposed by the instructor. For those exercises, the final answer is not what counts, but rather the class discussions they elicit.

Each instructor is left to his own devices to generate examinations. I have assigned projects based on Chapter 14 as a final exam. During the semester I give two examinations. For the first one, each student is required to go to the library and find an article that he or she thinks can be applied. After I approve of the article, the student is required to write a short paper (and give a 10 minute presentation) that demonstrates understanding and shows how he or she would apply the concepts at work. For the second examination, the students are required to write a software development plan for the final project.

Many of the ideas in this book were developed while I was at Planning Research Corporation (PRC). PRC is one of the largest suppliers of software services to the government. At PRC, I helped organize their systems engineering organization and managed one of its two departments. My group developed and implemented that company's software engineering methodology.

We produced their software standards and procedures and a software development plan that could be tailored to any company project. We also developed and taught a training course on software engineering. The primary contributors to those ideas were W. Barrie Wilkinson, Neil McDermott, C. Randy Allen, Charles Shartsis, and J. Kendrick Williams. Many of the ideas on knowledge data bases (in Chapter 10) were due to Charles Shartsis.

The encouragement and support of my partners at Avtec Systems, Inc., Ron Hirsch, Steve Mellman, and Jay Schwartz, are also acknowledged. Most of the material on fault tolerance in Chapter 12 was taken from a report by Jay Schwartz.

I thank my children, Reenie, Elisa, and Jeremy, for giving up so much of the time that they should have had with me. Their sacrifice is realized in subtle ways, like Jeremy's occasional question, "Daddy, when will you be finished with your story?" Most of all, I need to thank my wife, Madeline Zoberman Shere. Over two decades ago my research professor said to me, "I'm glad you married Maddy. She is good for you." I keep finding more reasons why he was correct.

SOFTWARE ENGINEERING AND MANAGEMENT

Contents

CHAPTER 1

Introduction

Software engineering is not a single process, instruction manual, or organization. It is the systematic use of many disciplines, tools, and resources for the practical application of computer hardware. In the 1930s, engineering was defined in Webster's (1936) as "the art and science of managing engines for practical application." If we consider our engines to be computers, this definition seems especially appropriate to software engineering.

This definition of software engineering is very broad; it includes almost everything except hardware. As we shall see during the discussion of the system development life cycle, in Chapter 3, many people consider software engineering to be those activities beginning with the analysis of software requirements. The viewpoint of this book is that software engineering begins with the system concept definition. Concept definition and system requirements have frequently been determined by hardware personnel with no input from software personnel. People performing these activities should include both software-oriented people and hardware-oriented people. They should also include expected system users. This team is needed to avoid imposing unrealistic performance requirements on either the hardware or the software.

To manage these engines, we have to consider all aspects of the intended application. These aspects include operational concepts, requirements, design, development, and maintenance. An aspect of managing engines that is generally ignored in the computer industry is developing criteria for determining when a system is no longer useful. That is, when should the system be replaced, and how do we retire the existing system?

Usually we consider this aspect of software engineering (and system engineering) only when developing a plan for transition from the existing system or procedures to the system we are going to develop. We don't plan for the death of the new system. Some people claim that systems don't die—they evolve. If that is the case, then why is it neces-

sary to spend billions of dollars to replace existing systems? In the case of the FAA Air Traffic Control System, the Government has committed an expenditure in excess of $1 billion dollars during a very tight budget period to redesign the system and to replace the hardware and software. This project was expected to cover an eight-year period from its inception in 1983.

This book is not a book on either structured programming or structured analysis. There are other books devoted to those subjects. Some of those books are cited in the bibliography. The subject of this book is software engineering. Software engineering is a superset of structured programming and structured analysis. For example, structured analysis techniques exclude from their scope cost analyses, simulations, legacy assessments, and a host of other analyses necessary for the design and development of a software system. This book describes what you need to do to plan, manage, and develop software systems and discusses some techniques for how to do it. The intent is to provide the reader with a system-level view of systems.

The system development life cycle is discussed in detail. Products associated with each phase of the life cycle are described. These descriptions are important because knowing the phase of the life cycle in which your task occurs and the products of that phase enables you to determine what you need to produce; your task becomes bounded. You may still need to determine how you will obtain the products, but you have eliminated the floundering time—time spent trying to figure out what to do.

In other engineering disciplines, there are handbooks, standards, and professional engineer examinations. The handbooks help us define our problems and serve as a reference for how to solve problems. The standards provide professional judgment on procedures. The professional engineer examination is used to certify that engineers are familiar with these handbooks and standards. Unfortunately, the software engineering discipline is just emerging. These handbooks, standards, and certification procedures do not yet exist for our industry, although some progress has been accomplished.

The greatest progress has been in the area of standards. The lead in developing and establishing standards has been the Department of Defense. There are extensive military standards related to software development. Many other agencies and many companies use these standards as the basis for their internal software standards and procedures. There are computer programmer examinations, but these are generally ignored. There is also a lack of conviction in the industry that these examinations reflect ability in software engineering.

Some companies have taken it upon themselves to develop their own standards and to "certify" their engineers by providing training courses. Significant portions of this book are based on the software methodology developed by and under the direction of the author for Planning Research Corporation (PRC), one of the largest suppliers of software services to the government.

In this introduction we discuss the need for a formal approach to software engineering, the applicability of a formal approach, and the institutionalization of a standard methodology. The reader who is working on small projects, who is a student, or who is not in a position to change the way his or her company does business should not jump to the conclusion that this introduction does not apply. This conclusion would be false. Company

operating committees may pontificate and pronounce policy, but it is the project managers, task leaders, and people who work for them that implement things.

A formal approach is presented to establish an attitude, a plan of attack, a way of thinking. That is what software engineering is all about.

1.1 DO I NEED A FORMAL SOFTWARE METHODOLOGY?

Why do we need a formal software engineering methodology? Either we can answer this question to the satisfaction of line management, key marketing personnel, and the vast majority of the technical staff, or we do not need a formal methodology.

To line management and marketing, the justification must be in terms of the bottom line, dollars and cents. For example, at PRC Government Information Systems the need was recognized by executive management. One of that company's strategic goals has been to move from a software company, with considerable work in facilities management and coding, to a systems engineering and integration company. As one step toward accomplishing this goal, a systems engineering group was formed. A primary objective of this group was to develop and implement a software methodology that would enhance the company's vitality and performance.

There was no intent for that group (or this book) to present a new structured technique. Over a half-dozen different techniques are discussed in subsequent sections. These are good enough, there is no need for another. The methodology presented here is a practical approach for managing, designing, building, and maintaining systems. It is based on things that are known to work because they have been used. Any structured technique can be used with this methodology.

Being able to use any structured technique is a requirement of methodologies used by companies performing software engineering for many clients. Some clients have very strong opinions about structured techniques. Some company field offices have been using specific structured techniques for many years; it would be foolish to tell them that they can no longer use these techniques. If you did, you could be assured that your new method would not be implemented.

Other companies or organizations are in a position to specify a single structured approach. Depending on the circumstances, it may also be advantageous to use a very formal, rigid approach. For example, an organization that runs a large software operation for a single corporation may insist that all documentation be of a specified format. A single approach to structured diagrams is important. This is also true of small companies producing business applications that must be tailored to the customer. Each applications programmer should use a similar approach to lessen confusion when reading another person's documentation.

As a first step toward developing a corporate methodology, a generic Software Development Plan (SDP, pronounced *ess dee pee*) that would be applicable to all software development should be produced. An SDP tells us what to do from a variety of perspectives. It includes a description of the development organization and schedule and an ex-

planation of the company's or organization's software development methodology. By nature, an SDP must be project specific to be completely useful. Thus, the generic SDP should include instructions for tailoring it to specific projects.

Next a generic Software Standards and Procedures (SSP, pronounced *ess ess pee*) should be produced. Sometimes the standards and procedures are included as part of the software development plan. When this is done, the combined document tends to be thick, so I recommend that these be written as two different documents. Writing the software standards and procedures could be done concurrently with the development of the SDP. Whereas the SDP tells us what to do, the SSP tells us how to do it. At PRC, these two documents have proved to be extraordinarily useful.

We had no trouble convincing top line and marketing management that our approach should be implemented. We provided internal consulting and temporary staffing for projects that had had problems. We delivered top-quality products to the delight of the client and line management. Applying the methodology paid off.

For marketing, we wrote the technical approach on proposals and provided easy-to-read documentation. Our approach won or helped win major contracts (valued in excess of $10 million each). In large companies working on systems development contracts, it is important to specify the technical approach in proposals. When the proposed work is won, the company is obligated to use the approach it espoused. This approach is key to changing the way a mid-sized or large company does business.

Some details of the implementation approach will be discussed later. The pace at which the software methodology was adopted throughout the company was only limited by the availability of personnel for training the staff and by the development of some of the tools needed to implement the methodology.

The line manager needs to be convinced that using the standardized approach increases the likelihood that his or her projects will be completed on schedule and within cost. Key marketing managers (who may be the same people as the line managers) have to be convinced that the formal technique will help sell jobs. These sales may be to other companies, to the government, or to other organizations within the company. Task leaders, analysts, designers, and programmers all need to be convinced that it is good for their careers.

In the training course on Software Engineering mentioned previously, the trainees were asked to specify what they thought were the pros and cons of a formal methodology. Their answers are given in Table 1-1.

The primary objections were fears that the client would prefer some other design technique and that the approach might be too rigid. The latter fear was expressed by someone who had just taken a five-day course on a particular structured design approach. When he left the course, he had fourteen thick notebooks and did not have enough shelf space in his office to store them.

When they were told that the standardized approach given here could accommodate all the well-known structured techniques, there was enthusiastic endorsement. The reasons soon became clear. Everybody could speak a common language and could describe