SOFTWARE ENGINEERING CONCEPTS

Richard E. Fairley

73.87221
F172

SOFTWARE ENGINEERING CONCEPTS

Richard E. Fairley

School of Information Technology
Wang Institute of Graduate Studies
Tyngsboro, Massachusetts

87

McGraw-Hill Book Company

New York St. Louis San Francisco Auckland Bogotá Hamburg
Johannesburg London Madrid Mexico Montreal New Delhi
Panama Paris São Paulo Singapore Sydney Tokyo Toronto

)287/56

B1

This book was set in Times Roman by Interactive Composition Corporation.
The editors were Debbie K. Dennis and Sheila H. Gillams;
the production supervisor was Charles Hess.
The cover was designed by Suzanne Haldane.
The drawings were done by ECL Art.
R. R. Donnelley & Sons Company was printer and binder.

SOFTWARE ENGINEERING CONCEPTS

1234567890 DOCDOC 8987654

ISBN 0-07-019902-7

Library of Congress Cataloging in Publication Data

Fairley, R. E. (Richard E.), date
Software engineering concepts.

(McGraw-Hill series in software engineering and technology)
Includes bibliographical references and indexes.
1. Electronic digital computers—Programming.
2. Computer programs. I. Title. II. Series.
QA76.6.F35 1985 001.64'2 84-12210
ISBN 0-07-019902-7

PREFACE

During the past decade, increasing attention has been focused on the technology of computer software. As computing systems become more numerous, more complex, and more deeply embedded in modern society, the need for systematic approaches to software development and software maintenance becomes increasingly apparent. Software engineering is the field of study concerned with this emerging technology.

Primary goals for this text are to acquaint students with the basic concepts and major issues of software engineering, to describe current tools and techniques, and to provide a basis for evaluating new developments. Many different techniques are presented to illustrate basic concepts, but no single technique receives special attention. Individual instructors may choose to emphasize particular techniques, depending on local circumstances, the backgrounds of their students, and their own interests.

The text is written for juniors, seniors, graduate students, and practitioners of software engineering. Those with more experience in the software field will profit more from the material. Minimal preparation for the material includes a course in data structures and exposure to system software concepts.

The layout of the text follows the traditional software life cycle: an introductory chapter is followed by chapters on planning, cost estimation, and requirements definition. These are followed by chapters on design, implementation issues, and modern programming languages. Chapters on quality assessment and software maintenance conclude the text. This layout should not be taken as an indication that the phased life-cycle approach to software development is the only method presented in the text. The use of prototypes and successive versions is emphasized throughout the text and in the term project material, which is presented in the appendix.

It is strongly recommended that a course in software engineering incorporate a term project. The project should involve team participation and preparation of various software engineering documents. This provides a focal point for the course and allows students to practice various techniques in the different phases of the project. The appendix provides numerous project suggestions, a schedule of project milestones, and formats for the project documents.

During the semester project, teams of three or four students per team prepare a requirements definition, architectural and detailed design specifications, a test plan, a user's manual, and documented source code for their software products. Because they are working against fixed deadlines imposed by the semester calendar, students are required to define three versions of their systems: a prototype, a modest version, and an enhanced version. Versions are defined so that functionality and performance characteristics can be moved between the versions as necessary. This allows a team to meet or exceed project milestones by adding or removing features in the current version of their system. Students are thus confronted with the realities of project deadlines, and they are also able to produce operational software products by semester's end.

Some teams will choose projects that are too ambitious and produce only a prototype. Others will produce all three versions of their product. In any case, it is important that students experience the frustration and satisfaction of integrating their subsystems into well-designed, well-documented functioning software products. If the project is not carried through to implementation, students may not realize the benefits of their analysis, design, test planning, and documentation efforts. Hence, the importance of defining successive versions of their systems. I have found this approach of successive versions to be valuable on "real" software projects that must meet fixed deadlines, and the documents prepared by the students are typical of good software engineering practice. Students thus learn valuable techniques in the process of learning software engineering concepts.

There are two ways to present the material in this text: in the sequence as written, or by first covering Chapters 6, 7, and 8 (programming techniques, programming languages, and quality assessment), followed by Chapters 4 and 5 (requirements definition and software design), followed by Chapters 2 and 3 (planning and cost estimation), followed by Chapter 9 (software maintenance). The latter approach has the advantage of building on the student's knowledge, and terms such as data abstraction and information hiding can be illustrated in an implementation language at the beginning of the course; however, a different approach to the term project is required. In this case, students are given an existing software product to critique and test during the early part of the course. During the latter part of the course they are required to analyze, design, and implement a significant modification to the product.

Both approaches have been used and found to be satisfactory. The author's preference is to present the material in the sequence as written, and to provide implementation language illustrations of terminology as it is introduced. This allows students to define and develop their own term projects following the schedule presented in the appendix. In presenting the material in sequence, the instructor often discusses software cost estimation (Chapter 3) following design (Chapter 5), or at the end of the semester in conjunction with software maintenance (Chapter 9).

I am indebted to the reviewers, namely, Bill Riddle, James Redmond. Tony Wasserman, and Marv Zelkowitz. They found time in their busy schedules to read the text and offer valuable suggestions for improvement. Also, thanks to series editor Peter Freeman for his encouragement and counsel, and thanks to Janice Fairley, who typed the first draft of the manuscript.

I owe a particular debt of gratitude to my students, from whom I have received more than I have given, and to the many researchers and practitioners of software engineering whose ideas and techniques are reported here. I have attempted to acknowledge their contributions where possible. I apologize for any oversights or inaccuracies in my acknowledgments.

Finally, I want to acknowledge the understanding and support of my wife, Janice; she makes it all worthwhile.

Richard E. Fairley

CONTENTS

CHAPTER
ONE

INTRODUCTION TO SOFTWARE ENGINEERING

INTRODUCTION

This text is concerned with development and maintenance of software products for digital computers. A software product, in contrast to software developed for personal use, has multiple users and often has multiple developers and maintainers. In most cases, the developers, users, and maintainers are distinct entities. Development and maintenance of software products thus requires a more systematic approach than is necessary for personal software.

In order to develop a software product, user needs and constraints must be determined and explicitly stated; the product must be designed to accommodate implementors, users, and maintainers; the source code must be carefully implemented and thoroughly tested; and supporting documents such as the principles of operation, the user's manual, installation instructions, training aids, and maintenance documents must be prepared. Software maintenance tasks include analysis of change requests, redesign and modification of the source code, thorough testing of the modified code, updating of documents and documentation to reflect the changes, and distribution of the modified work products to appropriate user sites.

The need for systematic approaches to development and maintenance of computer software products became apparent in the 1960s. During that decade, third-generation computing hardware was invented, and the software techniques of multiprogramming and time-sharing were developed. These capabilities provided the technology for implementation of interactive, multiuser, on-line, and real-time computing systems. New applications of computers based on the new technology included systems for airline reservations, medical information, general-purpose time-sharing, process control, navigational guidance, and military command and control.

Many such systems were built and delivered, and some of those attempted were never delivered. Among those systems delivered, many were subject to cost overruns, late delivery, lack of reliability, inefficiency, and lack of user acceptance. As computing systems became larger and more complex, it became apparent that the demand for computer software was growing faster than our ability to produce and maintain it.

A workshop was held in Garmisch, West Germany, in 1968 to consider the growing problems of software technology. That workshop, and a subsequent one held in Rome, Italy, in 1969, stimulated widespread interest in the technical and managerial processes used to develop and maintain computer software (NAU76). The term "software engineering" was first used as a provocative theme for those workshops.

Since 1968, the applications of digital computers have become increasingly diverse, complex, and critical to modern society. As a result, the field of software engineering has evolved into a technological discipline of considerable importance.

According to Boehm (BOE76a), software engineering involves "the practical application of scientific knowledge to the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them." As Boehm points out, the term "design" must be interpreted broadly to include activities such as software requirements analysis and redesign during software modification. The IEEE *Standard Glossary of Software Engineering* terminology (IEE83) defines software engineering as: "The systematic approach to the development, operation, maintenance, and retirement of software," where "software" is defined as: "Computer programs, procedures, rules, and possibly associated documentation and data pertaining to the operation of a computer system."

In this text, we use the following definition:

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

The primary goals of software engineering are to improve the quality of software products and to increase the productivity and job satisfaction of software engineers.

Software engineering is a new technological discipline distinct from, but based on the foundations of, computer science, management science, economics, communication skills, and the engineering approach to problem solving.

Software engineering is a pragmatic discipline that relies on computer science to provide scientific foundations in the same way that traditional engineering disciplines such as electrical engineering and chemical engineering rely on physics and chemistry. Software engineering, being a labor-intensive activity, requires both technical skill and managerial control. Management science provides the foundations for software project management. Computing systems must be developed and maintained on time and within cost estimates; economics provides the foundation for resource estimation and cost control. Software engineering activities occur within an organizational context, and a high degree of communication is required among customers, managers, software engineers, hardware engineers, and other technologists. Good oral, written, and interpersonal communication skills are crucial for the software engineer.

Because software engineering is concerned with development and maintenance of technological products, problem-solving techniques common to all engineering disciplines are utilized. Engineering problem-solving techniques provide the basis for project planning, project management, systematic analysis, methodical design, careful fabrication, extensive validation, and ongoing maintenance activities. Appropriate notations, tools, and techniques are applied in each of these areas. Furthermore, engineers balance scientific principles, economic issues, and social concerns in a pragmatic manner when solving problems and developing technological products. Concepts from computer science, management science, economics, and communication skills are combined within the framework of engineering problem solving. The result is software engineering.

Software engineering and traditional engineering disciplines share the pragmatic approach to development and maintenance of technological artifacts. There are, however, significant differences between software engineering and traditional engineering. The fundamental sources of these differences are the lack of physical laws for software, the lack of product visibility, and obscurity in the interfaces between software modules.

Software is intangible: it has no mass, no volume, no color, no odor—no physical properties. Source code is merely a static image of a computer program, and while the effects produced by a program are often observable, the program itself is not. Software does not degrade with time as hardware does. Software failures are caused by design and implementation errors, not by degradation. Because software is intangible, extraordinary measures must be taken to determine the status of a software product in development. It is easy for optimistic individuals to state that the product is "95 percent complete" and difficult for software engineers and their managers to assess progress and spot problem areas, except in hindsight. Many of the concepts discussed in this text are concerned with improving the visibility of software products.

Because software has no physical properties, it is not subject to the laws of gravity or the laws of electrodynamics. There are no Newton's laws or Maxwell's equations to guide software development. Intangibility and lack of physical properties for software limit the number of fundamental guidelines and basic constraints available to shape the design and implementation of a software product. Software design is comparable to architectural design of buildings in the absence of gravity. Excessive degrees of freedom are both a blessing and a curse for software engineers.

In a very real sense, the software engineer creates models of physical situations in software. The mapping between the model and the reality being modeled has been called the intellectual distance between the problem and a computerized solution to the problem (DIJ72). A fundamental principle of software engineering is to design software products that minimize the intellectual distance between problem and solution; however, the variety of approaches to software development is limited only by the creativity and ingenuity of the programmer. Often it is not clear which approach will minimize the intellectual distance, and often different approaches will minimize different dimensions of the intellectual distance. This discussion is not meant to imply that programming is entirely ad hoc, or that there are no fundamental principles of software engineering; however, the principles and guidelines must always be tempered by the particular situation.

Obscurity in the interfaces between software modules also distinguishes software engineering from the traditional engineering disciplines. A fundamental principle for managing complexity is to decompose a large system into smaller, more manageable subunits with well-defined interfaces. This approach of divide and conquer is routinely used in the engineering disciplines, in architecture, and in other disciplines that involve analysis and synthesis of complex artifacts. In software engineering, the units of decomposition are called "modules." (A precise definition of "software module" is presented in Chapter 5.)

Software modules have both control and data interfaces. Control interfaces are established by the calling relationships among modules, and data interfaces are manifest in the parameters passed between modules as well as in the global data items shared among modules. It is difficult to design a software system so that all the control and data interfaces among modules are explicit, and so that the modules do not interact to produce unexpected side effects when they invoke one another. Unexpected side effects complicate the interfaces between modules and complicate documentation, verification, testing, and modification of a software product. Programmers who intentionally write convoluted programs that have obscure side effects are known as hackers. "Hacker" is not a complimentary term in software engineering.

Finally, the data interfaces between modules must be exact. For instance, the number and types of positional parameters passed between routines must agree in every detail. There is no concept of an "almost" integer parameter or a global array of "almost" proper dimension. Constructing a stable software system is analogous to constructing a skyscraper with the stability of the entire structure depending on an exact fit of every door on every janitor's closet in every subbasement of the building; less than exact fit might cause the entire structure to crash.

During the past decade significant advances have occurred in all areas of software engineering: analysis techniques for determining software requirements and notations for expressing those requirements have been developed; methodical approaches to software design have evolved and design notations have proliferated; implementation techniques have been improved and new programming languages have been developed; software validation techniques have been examined and quality assurance procedures have been instituted; formal techniques for verifying software properties have evolved; and software maintenance procedures have been improved. Management techniques have been tailored to software engineering, and the problems of group dynamics and project communication have been explored. Quantitative models for cost estimation and product reliability have evolved, and fundamental principles of analysis, design, implementation, and testing have been discovered. Automated software tools have been developed to increase software quality, programmer productivity, and management control of software projects. New technical journals have been created, and existing journals devote increasing attention to software engineering topics. International conferences are held on a regular basis.

All this activity should not be interpreted to mean that the problems of software engineering have been solved. In fact, the level of activity in software engineering is indicative of the vast number of problems to be solved. Every technology evolves through the predictable phases of ad hoc invention, development of systematic procedures, and eventual culmination in a routine handbook approach to the discipline. As you read this text, it will become apparent that the technology of software engineering still involves a great deal of ad hoc invention.

1.1 SOME DEFINITIONS

We have defined software engineering as the technological discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates. Software engineering differs from traditional computer programming in that engineering-like techniques are used to specify, design, implement, validate, and maintain software products within the time and budget constraints established for the project. In addition, software engineering is concerned with managerial issues that lie outside the domain of traditional programming. On small projects, perhaps involving one or two programmers for one or two months, the issues of concern are primarily technical in nature. On projects involving more programmers and longer time durations, management control is required to coordinate the technical activities.

In this text the term "programmer" is used to denote an individual who is concerned with the details of implementing, packaging, and modifying algorithms and data structures written in particular programming languages. Software engineers are additionally concerned with issues of analysis, design, verification and testing, documentation, software maintenance, and project management. A software engineer should have considerable skill and experience as a programmer in order to understand the problem areas, goals, and objectives of software engineering.

It is sometimes said that software engineering concepts are applicable only to large projects of long duration. On large projects standard practices and formal procedures are essential, and some of the notations, tools, and techniques of software engineering have been developed specifically for large projects. On a small project, one can be more casual, but the fundamental principles of systematic analysis, design, implementation, testing, and modification remain the same, whether for a one-person, one-month project or a 1000-person, 10-year project. The fundamental concepts of software development and maintenance presented in this text are useful on every programming project; however, a few of the techniques discussed are cost-effective only on large projects.

The term "computer software" is often taken to be synonymous with "computer