CASE:
Computer-Aided
Software Engineering

CASE: Computer-Aided Software Engineering

T. G. Lewis
Oregon State University

VAN NOSTRAND REINHOLD
New York

Library of Congress Catalog Number: 90-38844
ISBN 0-442-00361-7

Printed in the United States of America

Van Nostrand Reinhold
115 Fifth Avenue
New York, New York 10003

Chapman and Hall
2-6 Boundary Row
London, SE1 8HN, England

Thomas Nelson Australia
102 Dodds Street
South Melbourne 3205
Victoria, Australia

Nelson Canada
1120 Birchmount Road
Scarborough, Ontario M1K 5G4, Canada

16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Trademark acknowledgements: Anatool is a trademark of Advanced Logical Software, Inc.; FullPaint is a trademark of Ann Arbor Softworks, Inc.; Image Grabber is a trademark of Sabastian Software; ResEdit, Macintosh XL, Macintosh II, LaserWriter, and RMaker are trademarks of Apple Computer Corp.; MacWrite, MacPaint, and MacDraw are registered trademarks of Claris Corp.; MicroPlanner+ is a trademark of Micro Planning International; PowerTools is a trademark of ICONIX Software Engineering; Ready, Set, Go! is a trademark of Letraset USA; SuperPaint is a trademark of Silicon Beach Software, Inc.; THINK Pascal is a trademark of Symantec, Inc.; ThunderScan is a trademark of Thunderware, Inc.

Library of Congress Cataloging-in-Publication Data

Lewis, T. G. (Theodore Gyle), 1941-
CASE: computer-aided software engineering / T.G. Lewis
p. cm.
Includes index.
ISBN 0-442-00361-7
1. Computer-aided software engineering. I. Title.
QA76.758.L49 1991
005.1--dc20

Preface

This book is intended mainly for practitioners who manage, design, code, test, and market modern software products. In addition, this book can serve as the text for a first course on software engineering in either an undergraduate or graduate program at most American universities. The material is suitable for a two-quarter sequence or a one-semester course. Lectures should be accompanied by small programming tasks and a course project — the Development Project — assigned early in the course so students can work on the project in parallel with the lectures. The course is highly pragmatic, and informal, and introduces many software tools to the software development process — hence the CASE designation.

The goals of this book are:

- To explain software engineering from a practical point of view, with an emphasis on CASE tools.
- To give a historical perspective of the development of software engineering.
- To cover both technical and human issues of software engineering, because it is still largely a human-driven activity.
- To relate software development techniques, tools, projects, and team structures to the kind of group environment in which software engineers must work and succeed. Specific recommendations for organizing software development teams are provided to guide developers through a successful project.
- To illustrate the concepts of software engineering through the consistent and pervasive use of a "real" example. The example is CoCoPro, a commercially available cost estimation tool based on the CoCoMo model.

The nature of software development is surveyed in the first chapter. The following three chapters prepare the reader to develop a successful software product. Chapter two surveys the field of lifecycle cost estimating and describes CoCoPro, the development project used throughout the text as an illustration of the ideas. Chapter three discusses the organization of teams to clarify the roles of each participant and to show how typical software development teams are organized. Chapter four covers general design

issues the good designer must be aware of, and chapter five begins the journey through the development project. These first five chapters should be mastered before the development project is attempted.

The early stages of the software development lifecycle are explained in chapters six, and seven. The development project is defined in these chapters and the outline of an application is provided as a high-level design. Here is where front-end CASE tools, which enter directly into the production of a running program, will be applied.

Chapters eight and nine give details of code production. We introduce the notion of a *cliche* in programming and use this concept to introduce details of the Macintosh. While very specific and detailed, this approach shows what a practicing programmer does during coding. Chapter nine is devoted exclusively to implementation and includes more specific examples. Readers are frequently "protected" from such details in a formal course on software engineering, but the approach taken here gives readers a glimpse into the real world of software development, on a real machine.

Chapters ten and eleven cover both the practical and semiformal approaches to software verification and validation. Chapter ten shows how to do unit and system integration testing. Chapter eleven describes more mathematical means of verifying software components.

Chapter twelve generalizes the information previously supplied by attempting to quantify "complexity." While highly controversial, complexity metrics are beginning to be used in the form of automated CASE tools. Chapter thirteen shows how several complexity metrics can aid in software maintenance.

Chapter fourteen introduces the notion of rapid prototyping and shows how to produce running applications using certain radically productive CASE tools that are not yet widely accepted. Rapid prototyping violates the waterfall model of software development, and where it applies, improves programmer productivity by a factor of 100!

If this book is used in a classroom setting, development teams should be assigned by the instructor. Team size should not exceed five or six students each, with three as the ideal. Team members follow the methodology described in the text and develop a commercial quality software product, complete with documentation and user manual. The development project should be limited to 5,000 to 8,000 lines of source code and should perform some useful function. CoCoPro, an implementation of CoCoMo (Cost Construction Model), is used as the sample development project.

The text recommends the use of the Apple Macintosh™ computer as both the design and development machine. The development project is implemented on a Macintosh, for Macintosh computers, in full compliance with Macintosh user interface rules and guidelines. Therefore, a considerable amount of time and effort is devoted to study of this machine and its software.

Why the Macintosh? The Macintosh system is itself an example of good software design. Its user interface is the result of years of scientific study. Direct manipulation with the mouse radically alters the programmer's experience and forces changes in the principles of both design and coding style. The toolbox is one of the best examples of reusable software, and the parameterized use of toolbox routines through the "resource" file is innovative, yet consistent with maintainable software design.

We use the Macintosh Pascal language as our implementation language because there are several state-of-the-art implementations of it (including object-oriented versions), and because it has been extended to incorporate modularity, one of the most beneficial features of modern programming languages. Separately compiled modules are called *units* in Macintosh Pascal, and are similar to the Modula II concept of modules, and the Ada™ concept of packages. However, most students of computer science are familiar with Pascal and should not have to learn a new language to apply the principles described in this book.

We also use a number of commercially available and homegrown CASE tools. These are the main focus of the book, which makes this text different from many others. The graphical interface of the Macintosh has stimulated development of a number of innovative and powerful tools for programmers. These tools are described in some detail, but the best exposure is to use them in one of the labs that accompany each lecture portion of the course. Classroom software should be made available for check-out from a lab consultant. Several copies are recommended for each class of twenty students.

The author has made several of these CASE tools available to the reader. To obtain these tools, contact the author, directly. These programs are intended to be used as pedagogical devices and are made available as is. The author is not responsible for correctness or completeness.

To obtain the programs, the author may be contacted by mail at the following address: Ted Lewis, Department of Computer Science, Oregon State University, Corvallis, OR 97331-3902; or, by Email: lewis@cs.orst.edu.

Acknowledgments

I would like to thank my students for the feedback they provided over the past five years while I was developing this course. Special thanks go to the exceptional students who produced high-quality software projects that are used in this book to illustrate many ideas: MacMan by Abdullah Al-Dhelaan; CoCoPro by Sherry Yang, Kirt Winter, Bob Singh, Abdennour Moussoui, and Ab Van Etten; OSU by Jim Armstrong, Fred Handloser III, Sharada Bose, Sherry Yang, Shyang-Wen Chia, Mu Hong Lim, Jagannath Raghu, Muhammed Al-Mulhem, and Haesung Kim; GrabBag by Jorge Sanchez; Style by Al Lake; UniTool by Tom Sturtevant, Mu-hong Lim, and Anil Kumar Yadav; and Vigram by Chia-Chi Hsieh and Kritawan Kruatrachue. Sherry Yang was directly involved in a number of these tools, and worked diligently to produce many of the figures and examples used in the pages to follow.

Contents

1

What Is Software Engineering?

PREVIEW

In this introductory chapter we survey the evolution of software engineering from troubled practice to emerging discipline. The lessons learned during the 1960s and 1970s were applied during the 1980s and led to a new approach called Computer-Aided Software Engineering, or CASE.

CASE tools incorporate what software engineers know about both the artifacts and the processes of software engineering. The artifacts — the program listings, documentation, data, and resource files — are only the most obvious components of software engineering. The process — the procedures, rules-of-thumb, and interaction among team members — is much more difficult to quantify. Yet both artifact and process are evolving toward automated means of producing, maintaining, and distributing software products.

We survey what is known about artifacts — application programs, for example — and what has been discovered about process — design, testing, inspection technology, and early defect removal, for example — to come up with recommendations for the practicing programmer. These recommendations will be followed as the book unfolds.

THE AGE OF SOFTWARE ENGINEERING

If computers are the steam engines of the postindustrial revolution, then computer software is the steam. Software is that invisible, almost ethereal quantity that goes into every industrial control system, business information system, video game, communication network, and transportation system, as well as thousands of other systems that we depend on daily. Unlike the steam of the industrial revolution, the intellectual steam of software consists of both artifact and process.

Software as artifact is literature in a tangible form: program listings, diagrams, and various kinds of documentation. More rigorously, **software** *is the sum total of computer programs, procedures, rules, and associated documentation and data pertaining to the operation of a computer system*[1]. We will be principally concerned with the manufacture and

GROWTH OF SOFTWARE ENGINEERING[2]

Pre–1969. Software development is out of control because of cost overruns and failures, especially in operating systems development. The term *software engineering* was coined as the theme of the NATO-sponsored meetings in 1968 and 1969.

1969–1971. First principles were established through research into good programming practices. Advantages of top-down design, stepwise refinement, and modularity were recognized. New programming languages including Pascal; new group techniques including Chief Programmer Teams introduced.

1972–1973. Structured programming and notions of programming style emerge. GOTO controversy subsides. Awareness of total software lifecycle grows and management and development aids are proposed.

1974–1975. Reliability and quality assurance concerns give rise to systematic testing procedures, notions of formal program correctness, models of fault tolerance and total system reliability. Early analysis of actual allocation of software development effort and expense appears.

1976–1977. Requirements, specification, and design. Renewed attention on early development phases prior to coding. Abstraction and modular decomposition as design techniques; structure charts, metacode as design representations. Increasing efforts to integrate and validate successive development phases of the software lifecycle.

1978–1980. Dispersion, assimilation. Increased use of automated software development tools; development of software engineering courses. First principles of 1969–1971 era begin to find widespread use in software industry.

1980–1989. Rise of CASE and the software engineering workstation. Automated tools corresponding to each phase of the software lifecycle begin to appear on stand-alone workstations.

1990–beyond. Application of expert systems techniques to software engineering. Combination of software engineering workstation, expert systems, and automated techniques for software development to find widespread use in the software engineering industry.

delivery of both programs and documentation to a user of the system in the form of a **software product:** *a product designated for delivery to a user*[1]. We will also call the software product an application, which consists of the deliverables of a software product, but does not include test cases, internal documentation, and miscellaneous software tools used to develop an application.

Software as process is difficult to define in rigorous terms because contemporary software developers build software systems without a complete understanding of the "physics" of software development. This has not discouraged the practicing software developer any more than the lack of a theoretical understanding of Newtonian mechanics discouraged the builders of ancient civilizations. Rather than wait for a theory to explain the dynamic nature of software development, practitioners have collected a group of techniques that seem to work, and have adopted *a systematic approach to the development, operation, maintenance, and retirement of software*[1] called **software engineering**.

Software engineering, more than anything else, is the practical side of software as process. It is deeply concerned with the **software development process** — *the process by which user needs are translated into software requirements, software requirements are transformed into design, the design is implemented in code, and the code is tested, documented, and certified for operational use*[1].

The gradual growth of software engineering is evidence of the struggle to understand software as both artifact and process involving machines, humans, and ideas. Growth has been slow because of the intellectual difficulty of formulating "laws" of software development and because of the extreme high degree of craftsmanship required to build

SOCIETY AND SOFTWARE

The software industry plays a major role in the computer industry and in the competitiveness of nations. As a vivid example of the concern over software, FORTUNE Magazine (How To Break The Software Logjam, September 25, 1989, pp. 100–112) published an alarming article on the "software crisis" in America. Here are some statistics on cost and complexity of popular software systems:

Product	*Lines of Code*	*Effort (man-yr)*	*Cost ($million)*
Lotus 1-2-3 version 3.0	400K	263	22
Space Shuttle	25.6M	22,096	1,200
1989 Lincoln Continental	83.5K	35	1.8
CitiBank Teller Machine	780K	150	13.2
IBM Checkout Scanner	90K	58	3

What is the solution to the high cost of software? Both technical and social complexities govern the production of software.

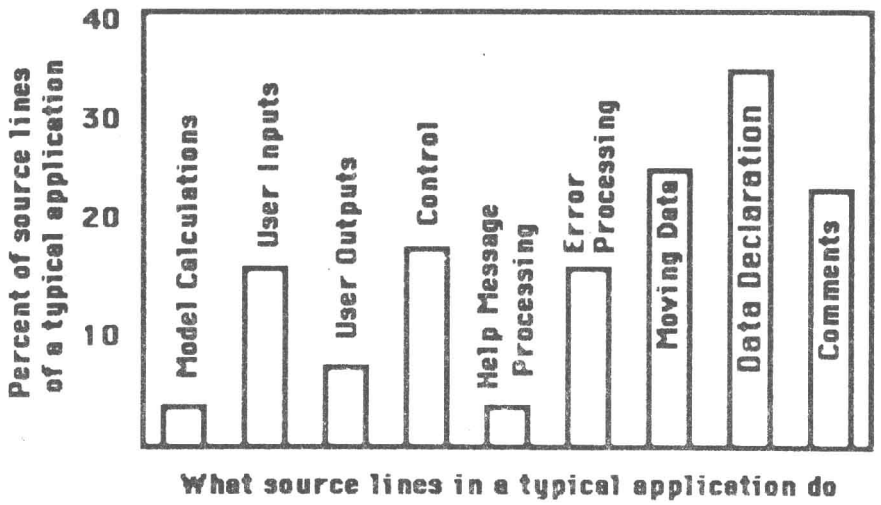
tools for software developers. It is clear, however, that such laws and tools are beginning to emerge in the form of theories and **automated tools** — *software tools that aid in the synthesis, analysis, modeling, or documentation of software*[1]. In the early 1980s these programs became known as CASE (Computer-Aided Software Engineering) tools. Hence the theme of this book: CASE tools in the form of simulators, analytic aids, design representations, documentation aids, and program generators provide the framework for the systematic study of software development.

We approach the study of software engineering through an understanding of artifact and process. First we examine the artifacts of software development, and then we look at the process itself. What is the nature of an application, and what are the factors that influence the process of software development?

THE NATURE OF AN APPLICATION

Applications differ from one another, but a typical application consists of source statements for doing the following:

- *Model Calculations:* Perform the calculations or operations intended by the application, e.g. payroll, stress, simulation, graphical, or database calculations.
- *User Inputs:* Interact with the user in order to capture the user's inputs. This may involve simple or complex interactions such as checking the input data for errors (bounds checking), and inserting the data into the program's data structures.
- *User Outputs:* Format and print or display the results of calculations, e.g. report writing.
- *Control:* Exert control in the form of comparisons, looping, and branching to carry out the logic of the program.
- *Help-Message Processing:* If the user requests help, display the appropriate help message and respond to user inquiry.
- *Error Processing:* In the event of an error during input, output, calculations, communications, etc. respond by displaying an error message, and then recover from the error.
- *Moving Data:* Move data from one data structure to another or from a database to the program's internal data structures. Sorting, searching, and formatting are data moving operations used to prepare the data for further processing.
- *Data Declaration:* Declare all data structures used by the application. For example, in Pascal, **const**, **type**, and **var** statements are used to declare all data structures.
- *Comments:* Provide clear, precise, and informative comments.

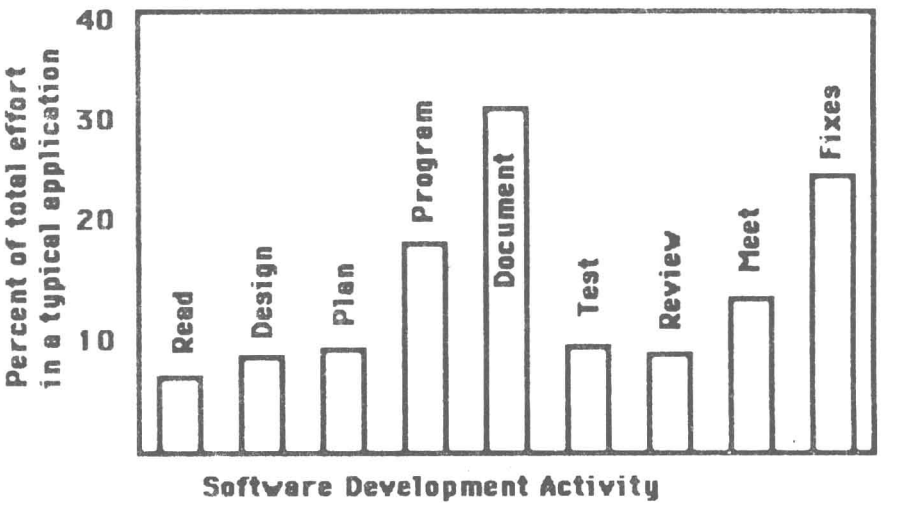


A typical application contains a very small percentage of statements devoted to the model calculations — a surprising observation given that this is the main purpose of the application. Notice that most of the program source statements involve data handling such as input/output, moving, and declaring data objects.

A glance at the list above might suggest an approach to developing an application: design and implement each of the parts, and then put them together into a single program. Unfortunately, because of the complexity implied in the terms of "design," "implement," and "put together" as we have used them here, software development is not so simple. **Complexity** — *the degree of complication of a system or system component*[1] — is determined by such factors as the number and intricacy of interfaces, the number and intricacy of branches, the degree of statement nesting, the types of data structures, and many other poorly understood characteristics of an application. These features of an application are missing from our list and are difficult to quantify. Hence, building an application is more than piecing together parts as the list above might suggest.

The complexity of software as artifact is responsible for "programmer productivity" difficulties. To understand the human side of programmer productivity, we need to understand what a programmer does when building an application. The activities of typical programmers in a typical project consist of the following:

- **Reading** about the system they are building and the tools and techniques they are going to use.



A typical software development project uses a programmer's talents in a variety of ways. Note, however, that actual programming takes up less than 20% of a typical programmer's effort. Documentation and fixing defects consume much more time than actual programming.

- **Designing** *is the process of defining the overall structure of the application, its components, modules, interfaces, and data structures, and then documenting the design*[1]. Design is not the same as programming, nor is it the same as program design. The design of an application involves the selection of data structures, algorithms, specification of information flows, as well as detailed program design.
- **Planning** *is describing an approach to be taken, the tasks to be performed, and the time schedules to be met.* Typically, a *WBS* (Work Breakdown Structure) is included in a plan that tells what is to be done, who is to do it, and when it is to be completed.
- **Programming** includes implementation of appropriate algorithms and data structures, commenting, and desk checking routines for correctness.
- **Producing documentation** — *any written or pictorial information describing, defining, specifying, reporting, or certifying activities, requirements, procedures, or results associated with programs, user manuals, and design, test, and modification documents.*
- **Testing** *is the process of exercising or evaluating a system component by manual or automated means to verify that it satisfies requirements or to identify differences between expected and actual results*[1]. Testing is not to be confused with debugging or defect removal. See Fixing for a description of defect removal.