SOFTWARE ENGINEERING IN THE UNIX®/C ENVIRONMENT

WILLIAM B. FRAKES
CHRISTOPHER J. FOX / BRIAN A. NEJMEH

AT&T

Software Engineering in the UNIX®/C Environment

William B. Frakes
Software Productivity Consori

Christopher J. Fox
AT&T Bell Laboratories

Brian A. Nejmeh
Instep Incorporated

Prentice Hall, Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging-in-Publication Data

Frakes, William B. (William Bruce)
Software engineering in the UNIX/C environment / William B.
Frakes, Christopher J. Fox, Brian A. Nejmeh.
p. cm. — (Prentice Hall software series)
Includes bibliographical references and index.
ISBN 0-13-829763-0
1. Software engineering. 2. UNIX (Computer operating system)
3. C (Computer program language) I. Fox, Christopher John.
II. Nejmeh, Brian A. III. Title. IV. Series.
QA76.758.F73 1991
005.1—dc20 90-24191
CIP

Editorial/production supervision: *bookworks*
Cover design: *Ben Santora*
Manufacturing buyer: *Kelly Behr and Susan Brunke*
Acquisitions editor: *Greg Doench*
Cover photo: *Courtesy Trustees of the National Maritime Museum.*
(A cosmographer at work. Drawn by J. Stradanus, late sixteenth century.)

Prentice Hall Software Series, Brian W. Kernighan, Advisor

Published by Prentice-Hall, Inc.
A Division of Simon & Schuster
Englewood Cliffs, New Jersey 07632

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

ISBN 0-13-829763-0

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 Pty, Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

For Gloria—Bill Frakes

For Susan—Chris Fox

For my wife and friend Laurie and our children
Mary Elizabeth and Emily Elise—Brian Nejmeh

Preface

This is a book about software engineering in the UNIX®* programming environment using the C programming language. There are already many books about software engineering, about the C language, and about the UNIX operating system. This book is different because it puts the tools and techniques offered by the UNIX/C environment into the context of a software engineering life cycle. This approach, based on software engineering courses we have taught at Rutgers and Columbia, has several advantages. First, it helps to make clear how these tools and techniques contribute to the overall goals of a software engineering project. Second, it clarifies the relationships among the tools and techniques. Finally, it shows which parts of the life cycle have strong tool support and which do not.

In selecting texts for software engineering courses, we found that many failed to provide examples of life-cycle products such as requirements documents and design documents. Students using these texts were often confused when given the task of creating life-cycle products for class projects, and we have been asked by practitioners for examples of life-cycle products as well. In this book, we present life-cycle products we created in building **ccount**, a small C metrics tool.

One question a reader of a book like this one might ask is: How much of the large and diverse field of software engineering does the book cover? This book is aimed primarily at those software engineers responsible for generating, testing, and documenting elegant, reliable, efficient, and maintainable code. Our bias, therefore, is strongly toward the *technical* rather than the *managerial* aspects of software engineering, a distinction we make as follows:

*UNIX is a registered trademark of AT&T. Hereafter we will use the abbreviation UNIX/C to stand for UNIX system C programming language.

- The managerial side of software engineering is concerned with project planning and scheduling, software cost estimation, project monitoring, project organization, and staff management.
- The technical side of software engineering is concerned with software design and implementation, software testing and quality assurance, and software maintenance.

In addressing the technical side of software engineering, we do not mean to belittle or slight software engineering managers. In fact, we believe that managing a software project is among the most difficult and important of all software engineering tasks. There is more than enough to be said about the technical aspects of software engineering in the UNIX/C environment to fill this book, however. In addition, managerial practices are inherently more generic than technical practices; managing a project in the UNIX/C environment is not very different from managing a project in other programming environments. Consequently we have little to say about software project management not already discussed in other excellent books.[1] [2] [3] [4] [5] [6] [7]

We hope that this book will be useful as both a tutorial and a reference for anyone building systems in C under some version of the UNIX operating system. The book should be of interest to designers, C programmers, and testers. Despite its technical orientation, this book should also be useful to managers responsible for products developed in the UNIX/C environment because it provides a catalog of the methods, tools, techniques, and practices available to their staff. This book should also be useful to students of computer science and engineering studying software engineering.

In our discussion we assume that our readers know how to program, and are at least novice users of the C programming language and the UNIX operating system. We do not assume any familiarity with software engineering, although, again, our survey of the field purposely neglects management issues, so our discussion of software engineering is incomplete.

One decision we faced in writing this book was whether to discuss UNIX tools that are not generally available. We decided to discuss the best tools we know of for UNIX/C development regardless of their availability. These tools demonstrate technologies to solve certain problems; a reader familiar with a technique for solving a problem can build the tool for himself or herself if necessary. For similar reasons, we also discuss needed tools that seem feasible but do not currently exist.

ACKNOWLEDGMENTS

Many people have contributed to this book. We would like to thank the following at Bell Laboratories: Tom Beattie, Susan Crocker, Nancy Firestone, Ed Fuchs, Judith Grass, Jeff Hooper, Brian Kernighan, Jim Krist, David Lubinsky, and Dave Neal. Our thanks to Steve Simmons at SUN Microsystems and P. J. Plauger of Plauger Associates for their helpful reviews. A special thanks to all our students at Rutgers and Columbia, especially Fritz Henglein, Maria Leone, and Bud Moore.

Chris Fox
Bill Frakes
Brian Nejmeh

REFERENCES

1. Boehm, B., *Software Engineering Economics.* Englewood Cliffs, N.J.: Prentice Hall, 1981.

2. Brooks, F. P., *The Mythical Man-Month.* Reading, Mass.: Addison Wesley, 1975.

3. DeMarco, T., *Controlling Software Projects: Management, Measurement and Estimation.* Englewood Cliffs, N.J.: Yourdon, 1982.

4. DeMarco, T., and T. Lister, *Peopleware: Productive Projects and Teams.* New York: Dorset House, 1987.

5. Fairley, R. E., *Software Engineering Concepts.* New York: McGraw-Hill, 1985.

6. Pressman, R. S., *Software Engineering: A Practitioner's Approach.* New York: McGraw-Hill, 1987.

7. Vick, C. R., and C. V. Ramamoorthy (eds.), *Handbook of Software Engineering.* New York: Van Nostrand, 1984.

Contents

1

Introduction

1.1 SOFTWARE ENGINEERING

On 15 January, 1990, AT&T's nationwide long-distance network was crippled for nine hours by a software fault. Millions of calls were blocked. Businesses such as travel agencies that depend on telephone service were virtually shut down. This was yet another demonstration of the importance of software in our lives and the sensitivity of our society to software errors.

The discipline that deals with such problems is called *software engineering*. Software engineering has been a recognized discipline only since the late 1960s. Still in its infancy, software engineering lacks a firm scientific basis. Indeed, software engineers disagree about the definition of the field itself. We define software engineering as the technical and managerial discipline concerned with the systematic invention, production, and maintenance of high-quality software systems, delivered on time, at minimum cost.

Software engineering has borrowed from many fields including computer science, mathematics, economics, and management theory. We focus on one part of this broad field—how to use a programming environment to produce high-quality software systems. Specifically, we discuss how to write such systems in the UNIX®* System C language environment. Our focus is thus on the technical rather than the managerial side of software engineering.

*UNIX is a registered trademark of AT&T.

1.2 UNIX SYSTEM AND C LANGUAGE

UNIX is a time-shared operating system originally written by Ken Thompson of Bell Laboratories in 1969. UNIX has two main parts. The UNIX kernel is a set of frequently used functions that are kept in main memory. The kernel schedules jobs, controls hardware, and manages input and output. The UNIX shell interprets commands sent to the system. "UNIX" is also often used to denote the set of tools and utilities—editors, compilers, software engineering support tools, and so forth—that typically come with the system. Many versions of UNIX exist. The best known are AT&T UNIX System V, and Berkeley UNIX—officially known as UCB 4.xBSD. Versions of UNIX are similar, but differ in their implementations and in the tools and utilities they provide. For example, several UNIX shells are available: the Bourne shell, the C shell,[1] and (our favorite) the Korn Shell.[2] All these shells can be used with different kernels.

The C language was written in 1970 by Dennis Ritchie. Thompson and Ritchie rewrote the UNIX kernel in C in the early 1970s, and since then, UNIX and C have been linked. C was originally written as an alternative to assembly language for systems programming. As such, it allows a programmer great freedom—it is lax, for example, in its handling of data types. It also allows low-level access to the machine. C is known for producing fast, efficient code.

C has become enormously popular. Besides systems programming, C is used to build large-application software systems. The latest generation of large switching systems at Bell Laboratories, containing millions of lines of code, for example, is written in C. Whether C is a good language for large-system software engineering is the subject of heated, often religious, debate. Until enough empirical data are gathered to answer this question, all we can say is that it has been used successfully to build large systems.

One indication of the popularity of UNIX/C is the many books about them. These books describe both the C language and the UNIX system, and give detailed guidance to their use. In this book we do *not* provide detailed guidance to the use of the tools and techniques we discuss. Our purpose is to put UNIX/C in a large software engineering system context—to show how UNIX/C can be used throughout the life cycle to support a software engineering project.

1.3 STATE OF THE ART IN SOFTWARE ENGINEERING

Although many important technical contributions have been made to software engineering (for example, the development of high-level programming languages) the state of the art is far from what software engineers would like. Common practice is worse. Software development projects often have low productivity, and software products are often full of faults and do not meet user needs. To illustrate the extent of the problem, consider the findings of a study of nine Department of Defense software development contracts totaling \$6.8 million:[3]

- On software that was delivered but never successfully used, \$3.2 million was spent.
- On software that was paid for but not delivered, \$1.95 million was spent.

- On software that was delivered and used, but had to be extensively reworked or later abandoned, $1.3 million was spent.
- Out of the $6.8 million, $119,000 was spent on software that was used as delivered.

Unfortunately, such waste is common. Most large technical organizations can chronicle legendary software disasters.

Although software project failures are often attributed mainly to managerial problems, technical sources of waste and inefficiency contribute to project failures, and certainly contribute to high costs. For example, one well-known source of waste attributable in part to technical problems is failure to reuse software. DeMarco estimates that the average project reuses only 5 percent of code, despite evidence that much more code could be reused.[4] Technical problems about how to design, catalog, store, and retrieve reusable software components are not yet solved. One reason the UNIX/C software development environment is important for software engineering is that UNIX contains many small reusable tools. Reusable function libraries are a staple of the C language. We discuss these issues in greater detail in Chapter 6.

What can be done about the poor state of software engineering? One approach is better education about the problems of software engineering, and about the best available tools and techniques to solve them. This task is not adequately addressed by academic and industrial courses in programming and software engineering. In our experience doing and teaching software engineering, hiring and supervising software engineers, and interacting with developers on many projects, we have observed a widespread lack of knowledge about software engineering problems, and a host of bad practices. Some unfortunately common observations follow:

- Managers with no background in software engineering responsible for technical work in major software projects.
- Employees with little software engineering experience responsible for difficult technical tasks, such as the design or implementation of major portions of software systems, with inadequate technical training and guidance.
- Graduates of computer science programs at major universities who have never heard of software engineering, let alone the tools and techniques for producing high-quality software products. Many computer science programs do not offer courses in software engineering, or if they do, the courses are optional and focus on programming.

1.4 SOFTWARE PRODUCTS, PROJECTS, AND METHODOLOGIES

Software is some executable object such as source code, object code, or a complete program. A *software product* is software plus all the supporting items and services that together meet a user's needs.† A software product has many parts including manuals,

†Some authors refer to what we have called "software products" as simply "software." We prefer our terminology because it more closely reflects common usage.

references, tutorials, installation instructions, sample data, educational services, technical support services, and so forth. Software engineers produce software products, not just software.

Anything produced by a *software project* is a *work product.* Work products include (1) engineering documents used to define, control, and monitor the work effort; (2) executable objects like prototypes, test harnesses, and special purpose development tools; and (3) data used for testing, project tracking, and so forth. Software engineers help produce most work products because they have technical content. In fact,' software engineers often spend more time working on nonsoftware work products, especially documents, than they do working on software." ~ [illegible]

1.5 SOFTWARE PROJECT SIZE AND TYPE

Software projects come in many sizes. One way to classify them is by lines of code[‡] as in Table 1.1.

Table 1.1: Project Size Categories

Category	Programmers	Duration	Lines of Code	Example
Trivial	1	0–4 weeks	<1K	Sort utility
Small	1	1–6 months	1K–3K	Function library
Medium	2–5	0.5–2 years	3K–50K	Production C compiler
Large	5–20	2–3 years	50K–100K	Small operating system
Very large	100–1000	4–5 years	100K–1M	Large operating system
Gigantic	1000–5000	5–10 years	>1M	Switching systems

The largest software projects employ thousands of programmers, managers, and support personnel. System files and functions number in the tens of thousands and may be distributed across many machines. Changes made to one file may affect hundreds of others—and all the people who work on them. It is the complexity of the interrelationships among all these system elements that distinguishes software engineering in the large. It is difficult for someone who has not worked in a large project environment to appreciate this complexity. This is one barrier to teaching software engineering. People familiar with software engineering techniques for small projects may think that these techniques will scale up to large projects. This is usually not so. Software engineering tools, for example, sometimes do not work for large or distributed systems. Informal change management techniques adequate for a group of five developers will be disastrous for a group of fifty.

[‡]A *line of code* is a common measure of program size, but there is not a standard definition of what a line of code is. In this book, a line of code is a source code file line containing at least one language token outside a comment.

Software engineering practices are important for projects of every size. For large, very large, or gigantic projects, they are indispensable, because systems of such size could not be built without them. There is empirical evidence that project size has a major effect on important project attributes, such as individual programmer productivity, which decreases exponentially as system size and development team size increase. The reason for this effect is probably the need for more coordination and communication on a big project. Conte et. al.[5] provide a good discussion of empirical studies of factors affecting software projects.

Another factor that varies with project size is the amount of required project documentation. A trivial project, say a simple source code metrics program like **ccount**, may not need any engineering documents, and no more user documentation than a manual page. In contrast, a medium-sized project, such as a C compiler, should have a set of engineering documents that includes at least a concept exploration and feasibility document, a requirements document, a project plan, design documents, a test plan, and a project summary. The user needs at least a manual, and typically also tutorials, quick reference cards, installation instructions, and so forth. It would be wasteful (and discouraging to the developer) to require as much documentation for the metrics utility as for the C compiler. Nevertheless, such demands are sometimes made. This is a common example of software engineering practices inappropriately applied.

Projects differ in type as well. The performance requirements, designs, implementation strategies, testing methods, and problems encountered differ substantially for operating system programs, scientific application programs, business application programs, and embedded real-time systems. Productivity differences between different types of software projects have frequently been observed.[5] Software engineering practices must be adapted to projects in different domains.

The set of tools, techniques, and methods used by software engineers in a software project is called a *project methodology*. Choosing the right methodology for a project is difficult. The lead software engineer must form a project methodology by selecting from the available tools, techniques, and methods those appropriate for his or her project.

1.6 SOFTWARE LIFE CYCLE AND LIFE-CYCLE MODELS

Project methodologies are applied within the context of a *software life cycle*—the series of developmental stages, called *phases*, through which a software product passes from initial conception through retirement from service. A *life-cycle model* is a representation of the software life cycle that may also include information flows, decision points, milestones, and so forth. We stress that a life-cycle model is only that: a model. No real project will behave exactly as specified by a life-cycle model, and the divergence may be large.

The phases of a life-cycle model may be *temporal phases*—forming a sequence in time—or *logical phases*—representing steps not forming a temporal sequence. For example, implementation logically precedes testing, but parts of the implementation and testing phases may occur simultaneously. Thus a life cycle model using logical phases may have an implementation phase before a testing phase, whereas a model using temporal phases may have these phases overlap. A life-cycle model may be used *prescriptively* to mandate life-cycle events or *descriptively* to record life-cycle events. Many software life-cycle models have been proposed.[6] Most models agree on the fundamental phases of the life cycle, but differ in terminology, emphasis, flexibility, and scope. A detailed prescriptive temporal model is useful in a project plan because it maps the project's intended course. Descriptive temporal models are useful in documenting the life cycle and analyzing a project when it is over.

The life-cycle model we use in this book is a general prescriptive logical model. It is general in that it presents only the logical sequence of life-cycle phases. Although it would be better to present a specific temporal model, no such model will fit all software projects. Like other parts of a software methodology, specific life-cycle models must be constructed for specific software projects.

Our model is a standard waterfall model[7] consisting of the following logical phases:

- *Concept exploration and feasibility analysis phase.* Identify a need to automate a process and analyze project feasibility.
- *Requirements specification phase.* Analyze and document system requirements. The requirements document must clearly state *what* the projected system will do, what elements the software product will have, and what characteristics the product elements must have.
- *Design phase.* Design the system and document the design. The design document specifies *how* to build a software system to satisfy the requirements.
- *Implementation phase.* Write the software.
- *Testing phase.* Exercise the software to verify that it satisfies its requirements.
- *Maintenance phase.* Following deployment of the software product, correct faults; change and enhance the system.

This model, like any other, only loosely represents how a project actually works (Figure 1.1). Too many unforeseen things happen on a software project for any model to be more than a general guide. The waterfall model is often criticized as having little to do with project realities. Despite this, it is still the model most often used on large projects. Furthermore, it is a useful pedagogical framework, which is why we adopt it here.

1.7 ATTRIBUTES OF SOFTWARE QUALITY

Quality is linked to meeting user needs. One way to link quality with user needs is to follow Juran[8] in defining quality as *fitness for use*. Juran distinguishes two aspects of fitness for use: the collection of product features that meet user needs and freedom from deficiencies. A product with a collection of features that enables it to meet a user's needs makes for customer satisfaction. Freedom from deficiencies avoids customer dissatisfaction. Together these aspects of a product make it fit for use, or of high quality.

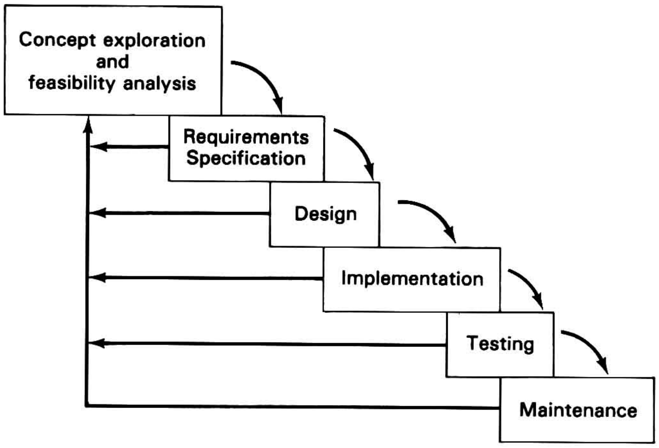
Concept exploration and feasibility analysis
Requirements Specification
Design
Implementation
Testing
Maintenance

Figure 1.1 Waterfall Model of Software Life Cycle.

The quality of finished software products depends largely on the quality of the work products generated during development and maintenance. The notion of quality as fitness for use yields attributes for evaluating the quality of work products based on the needs of the users of the work product. For example, requirements specification documents are used by customers and developers to record decisions and agreements about the product to be built, by designers as a definitive source of information about the product to be designed, by documenters as a source of information about how the product will behave and how it should be used, and by testers as a source of information about how the system should behave in response to test data, and about its performance parameters. These needs motivate specific quality attributes for requirements specification documents—for example, that all requirements be testable, precise, and clear; that performance constraints be explicitly stated; and so forth. Similar quality attributes can be generated for all work products; it turns out that most work products share a core set of quality attributes that includes the following:

- *Correct.* The definition of correctness varies. For example, a requirements document is correct if it accurately describes needed functions and properties of a product; software is correct when it meets its input-output requirements; program documentation is correct when it accurately describes a program.
- *Efficient.* This attribute refers to how well software uses computational resources. For example, quicksort is more efficient than bubblesort because it can sort a list with fewer machine instructions.
- *Maintainable.* This attribute can be applied to any work product but is most often applied to software. Maintainability is how easily a work product can be corrected, changed, or enhanced.
- *Portable.* Refers to how easily software can be moved to a variety of environments.
- *Readable.* This attribute applies to any textual work product. It refers to how easy it is for a person to read and understand the work product.