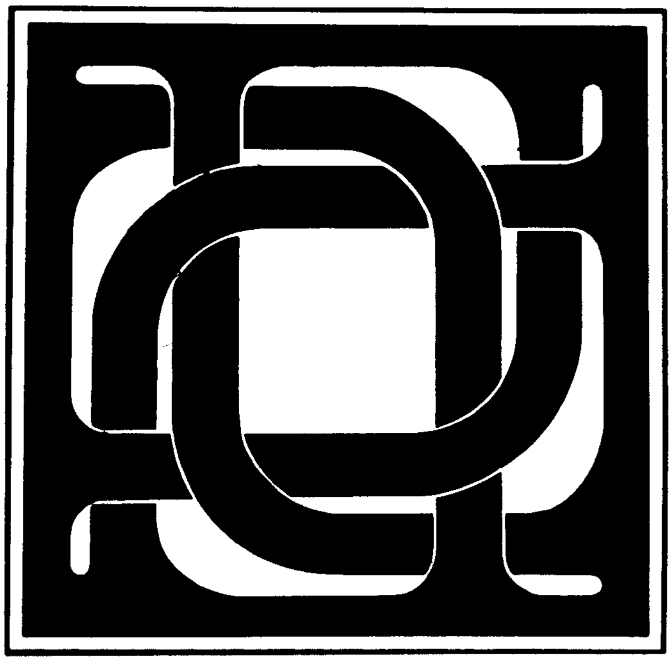
LAWRENCE J. PETERS

FOREWORD BY
L.A. BELADY



SOFTWARE DESIGN: METHODS & TECHNIQUES

LAWRENCE J. PETERS

FOREWORD BY
L.A. BELADY

SOFTWARE DESIGN: METHODS & TECHNIQUES

Yourdon Press
1133 Avenue of the Americas
New York, New York 10036

Library of Congress Cataloging in Publication Data

Peters, Lawrence J.
Software design.

Bibliography: p.
Includes index.
1. Electronic digital computers--Programming.
I. Title.
QA76.6.P473 001.64'25 80-50609
ISBN 0-917072-19-7

Printed in the United States of America

Library of Congress Catalog Number 80-50609

ISBN: 0-917072-19-7

This book was set in Times Roman by YOURDON Press, 1133 Avenue of the Americas, New York, N.Y., using a PDP-11/45 running under the UNIX[†] operating system.

[†]UNIX is a registered trademark of Bell Laboratories.

Acknowledgments

I wish to express my appreciation to my reviewers, whose patience and determination greatly improved this work. Specifically, I wish to thank

Kyu Lee, Seattle University
Harry Walker, Weyerhaeuser Company
Paul DuBose, Weyerhaeuser Company
Phil Milliman, Weyerhaeuser Company
Laszlo Belady, *IEEE Transactions on Software Engineering*

I would also like to thank all my colleagues whose comments, ideas, and complaints have been woven into the collage that is this book. My thanks to Seattle University's Software Engineering Graduate Program for letting me use a draft of this text in the Software Design class, and to the students whose suggestions greatly enhanced the resulting book.

Rick Eiber and Alice Wherrette, who produced the graphics, are also to be commended. Lorie Mayorga, Janice Wormington, and all of the people at YOURDON Press are extended sincerest appreciation for their suggestions and wisdom in knowing when and where to kick an author and when to leave him alone.

Finally, since we live in a world that is largely the result of our own work and not that of nature, I want to thank the world's designers, whose efforts have greatly contributed to our physical surroundings, our lives, and the way in which we perceive reality.

Preface

Our advances in software design are a direct result of our increased capacity to deal with systems as abstractions.

—L.P.

The field of software development has undergone some of its most profound changes in the last ten years. Much of this change has been in response to ever-increasing demands on software systems in terms of their complexity, reliability, and resiliency. Symptomatic of such rapid evolution is the proliferation of methods and techniques intended to solve "the" software problem. However, real-world software design problems often exhibit characteristics that make them unique. This forces the software engineer to seek alternative ways of composing and documenting a design.

Software development challenges the software engineer in several ways. Unlike many other fields, software systems will not be mass produced. This divorces the software engineer from many problems associated with manufacturing. However, since he is dealing with logic — the abstract — the results of his labor are difficult to identify with. Software operates on a time scale and reference frame that are incomprehensible by human standards. These factors, and the lack (for the most part) of an engineering background on the part of software developers, have led to the naive view that software design is unique and that its problems are exclusively those of software. The software engineer also has less guidance than technicians in other fields regarding the scope of his problem or the acceptability of his solution.

Software Design: Methods & Techniques is intended to meet the need for software design guidance, by describing both methods (strategies, recommendations, or guidelines based on a philosophical view) and techniques (tactics or well-advised "tricks of the trade"). It is directed at the professional software designer, novice, student, software manager, and customer. The software manager and customer can both use the book to get a concise description of the issues, benefits, and liabilities associated with using certain techniques or approaching certain software design problems in a given way, and they may also use it to increase communication. The others can utilize it as a resource guide providing several alternative methods and techniques; to aid experimentation with existing technology; and possibly to spark some new ideas, complement limited experience, and provide further insight regarding what software design is and the pros and cons of currently available techniques. The objective here is simply to cut through the mystique surrounding software design and its accompanying methodologies, leaving only the basics.

Software Design: Methods & Techniques is divided into four parts:

- Part I describes what design is, per se; what software engineering is; and how design manifests itself when its intended product is software.
- Part II surveys different schemes for representing various software design characteristics and discusses their effectiveness and compatibility.
- Part III surveys different methods for composing software designs and examines their effectiveness in specific design situations.
- Part IV describes an approach for composing software design methodologies tailored to specific project issues, and discusses some of the fundamental issues facing the software designer today.

Software Design: Methods & Techniques treats the subject of software design from several standpoints: its commonality with the problem of design in general, schemes for formulating and documenting designs, and case study guidelines. It presents several dozen techniques and demonstrates the use of each in sufficient detail to effectively use each technique. Appropriate references are provided should the reader require more detailed information. This book, with its guidance and examples, will prove an asset to the software designer, both for composing a software design methodology to accomplish a given task and for selecting a single technique to solve a specific problem.

Foreword

There are only two ways open to man for attaining a certain knowledge of truth: clear intuition and necessary deduction.

— René Descartes

Software Design: Methods & Techniques is on the *process* of mapping real-world phenomena onto computer programs. Since it has been written by an engineer with extensive experience in software design, this book reflects both the breadth of the process in question and the practical leaning of the book's author.

The notion of software design, and the perception of its importance, are relatively new. One used to talk simply about programming when a model, usually mathematical and already well prepared for a desk calculator, was transformed into a machine-executable procedure. This was a relatively small step as compared, for example, to the computerization of an entire banking operation, which is a set of activities never before formalized, whose functioning relies on well-trained personnel to coordinate routine tasks and solve unforeseen exception cases.

By the mid-1970s, the view that programming is but a fraction of the software development problem became well accepted. Since then, focus has turned to design, a complex process covering a variety of activities that must precede the act of writing a compilable program. At the same time, software people also discovered that designing *anything* — a car, a washing machine, furniture — is rarely a well-documented activity, is difficult to teach, and must often be based on apprenticeship at the master's knee, complemented later by hard-won experience.

Nevertheless, as Peters points out, design theorists agree that there are two major phases of any design process: diversification and convergence. Diversification is the *acquisition* of a repertoire of alternatives, the raw material for design: components, component solutions, and knowledge, all contained in catalogs, textbooks, and the mind. During convergence, the designer chooses and combines appropriate elements from this repertoire to meet the design objectives, as stated in the requirements document and as agreed to by the customer. This second phase is the gradual *elimination* of all but one particular configuration of components, and thus the creation of the final product.

Since this end product, by definition, is something that never before existed, it may contain inconsistencies. These are often impossible to predict at the time the design choice is made, but become manifest rather later in a larger context. This problem has two implications: First, that a design decision, when elaborated upon, could lead to an insight, which in turn may alter the original decision; in other words, design is inherently iterative. The second implication is that design methods are needed in order to recognize the occasionally unavoidable inconsistencies easily and early, so that no time is wasted pursuing a decision chain that will have to be scrapped later.

During his work, two distinct productivity issues occupy the designer's mind a great deal: the productivity of the system he designs, and his own productivity in creating it. The chosen design alternative could turn into a wasteful product, but optimizing it beyond a point may be too costly, drastically reducing design — or programming — productivity. One definitely needs methods facilitating the rapid discovery of product inefficiency at early stages of the design.

The need to discover inefficiency early makes it important to *externalize* (that is, make visible) an evolving design at each stage. Engineering blueprints, for instance, serve this purpose and are useful not only for a designer by calling his attention to trouble spots and potential inconsistencies, but also for a team or an entire organization developing a product: Blueprints are the major medium of communication, criticism, and collective refinement. Moreover, representation methods must be relatively simple and direct in bridging the gap between reality and the program; and they must be efficient during the multiple iterative steps.

Many of the methods in this book are recommendations as to which way, and in which order, the designer should proceed to model real-life data and their manipulation in his program. The methods are essentially different ways of representing software as its design evolves, guiding the decomposition of the whole into independently manageable pieces, enhancing communication within the design community, and helping to uncover inconsistencies early. *Software Design: Methods & Techniques* is comprehensive, offering an almost frightening variety of approaches. This proliferation is partly due to the nature of software: namely, that it is not a physical, tangible entity, and representing it is not as obvious as showing a piece of machinery by its orthogonal geometric projections. It is also partly due to the fact that software engineering is too young a discipline to have formulated its best methods.

Software products have been observed to evolve through a virtually never-ending series of modifications. Therefore, the accompanying design documents must be updated to reflect changes in the code. This is not equally easy with all of the many methods of representation, since some of them exist today only in hard-copy form, which requires an extra reproduction budget for each modified version. With progress in display technology, this problem may be alleviated in the future; yet today this leaves us with an unduly large variety of potentially useful, but economically not yet feasible, methods.

Finally, before I let the reader go ahead and enjoy the substance of the book, I would like to say a few words on the role of mathematics in the design process, which is defined differently by each of the cited approaches: I think its role should be mainly to aid the process of selecting from alternate arrangements and parameters, whenever key attributes — functional correctness, performance, resource demand — can be formally stated. For instance, in mechanical engineering, a tentative design must be completed as a basis for a formal verification of the integrity of the proposed structure. The result of the mathematical analysis may thus lead either to the acceptance of the designed alternative or to the study of another one. This appears to be the case in conventional engineering, and will likely remain valid in software engineering for a while, until breakthroughs in artificial intelligence may relieve us (or deprive us?) of the chore of designing software.

L.A. Belady
Senior Editor
IEEE Transactions on Software Engineering

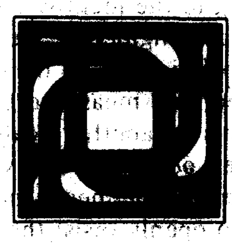
January 1981

Contents

PART I

Software: Engineering and Design

PART I
Software: Engineering and Design



Software design technology is a system — not a secret.

—L.P.

People have been writing computer programs for less than two generations. Compared to other engineering fields, the development of software is a newcomer to technology. But fledgling as it may be, it has probably had the greatest impact on industrial society since the invention of the steam engine. Devices such as the steam engine were labor savers for certain types of physical activities, and as such they had an effect on how we viewed problems of logistics, as well as on the types of problems we would choose to address. The computer's far broader scope of application makes it a unique asset in solving problems and evolving our view of the world.

Intuition might lead us to the conclusion that any field so important to other disciplines is, itself, a model discipline. Quite the contrary is true. Neither the subject of software engineering nor that of software design is a widely recognized discipline. The first graduate program in software engineering at an American university was launched only recently, with only a few other universities considering such action.* Worst of all, software engineers do not even know who they are! At the Third International Conference on Software Engineering, I asked an audience of more than 700 attendees how they had filled in the box marked "Occupation" on their income tax forms. By a show of hands, less than 10 percent indicated "software engineer" or, simply, "engineer." Clearly, the discipline still lacks recognition even within its own ranks. This situation is improving. At the Fifth International Conference on Software Engineering, I put the same query to an audience of about 500, and approximately 25 percent responded affirmatively — still not a majority, but a hopeful sign.

There are probably many contributing factors to the identity crisis suffered by software engineers. The two most prominent ones are the training of the people who are doing software engineering; and, second, the nature of the products they build. To-

*Seattle University in Seattle, Washington began offering a Master of Science in software engineering in the fall of 1979.

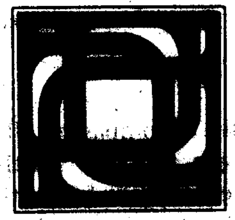
day, people who produce software come from many academic disciplines — not just science, mathematics, and engineering. Many software developers who lack an engineering background think of engineering as an exact discipline that produces formulated, precise, closed-form solutions to problems. The inexactitude associated with software design seems intolerable to many designers, who feel that if there were a true engineering discipline for software, all estimating and scheduling problems would go away. Actually, nothing could be further from the truth: Engineering depends as much or more on common practice and empirical knowledge as it does on scientific fact. Hence, one reason for the identity crisis is that many software engineers do not recognize what they are doing as engineering.

The second factor contributing to the identity crisis — the product — affects much more than the self-image of software engineers. In most other engineering fields, the product is something that can be experienced by the engineer. Its time frame and physical properties can be seen, touched, and measured by humans.

But software engineering deals in another realm. This fact has had a profound effect on our view of the activities of software development. Even today, there are many software designers who support the need for refinements and reviews in the design of physical systems, but who openly resist the use of these common engineering practices in software systems development, viewing such practices as contractual nuisances. The difference in reference frame can affect our perception of engineering. Before we can attempt a meaningful discussion of software design, we must establish a working understanding of software engineering and design.

In Part I, we will put software engineering and software design into perspective. First, we will identify the scope, content, and structure of software engineering. Then, we will examine software design from two standpoints: as it relates to the larger discipline of software engineering, and as a discipline in itself.

CHAPTER 1
The Role of Software Design in Software Engineering



Software engineering is at the interface between theory and practice.
—L.P.

Before we can begin to describe meaningfully what software design is, we need to understand how it relates to the other activities associated with the development (engineering) of software. Many problems connected with software development, particularly software design, are related to ignorance of the nature of the subject area and of its issues. A psychologist might call this phenomenon an identity crisis. Whatever you choose to call it, a crisis it very definitely is.

Today, more than ever, society relies on software, not computers. How secure would society be about the future if it were generally known that this key element of progress was being developed by people who had little formal training in the software crafts, had no accepted standards for practicing this science/art form, and did not even recognize the field in which they were operating?

One of the earliest uses of the term "software engineering" was in the naming of the first NATO Conference on Software Engineering in 1968 [1]. This conference and the introduction of the term grew out of concerns on the part of customers and software professionals alike about the cost and quality of the software being produced. These concerns prompted the adoption of many methods and techniques, such as top-down design, each promising to remedy some symptoms of the perceived problem. These techniques, however, were not based on the application of an engineering discipline to the production of software.

Although we have come a long way toward remedying these cost and quality problems, the tough part of the journey is still ahead. No longer are we faced with technical problems that can be quickly remedied with reasonably obvious solutions (using design and code reviews or restricting the use of GOTOs, for example). Today's solutions are much more subtle, oriented toward remedying problems associated with the production of systems, not individual programs.

The term software engineering is used today to describe a loosely coupled collection of practices, techniques, and methods. More diverse than most other engineering fields, it includes activities ranging from the conceptualization of software systems through their implementation and delivery. Mechanical engineers, for example, may design an aircraft component, but they are not responsible for producing the component. Manufacturing engineers and industrial engineers work on the latter task. In software, by contrast, the same engineers often design and produce a component.

We cannot hope to cure all of the ills of the industry in this book, but we can develop a framework and perspective with which to treat more effectively the subject of the remainder of this text. Specifically, the components of software engineering are organized into a model of the subject area in Section 1.3. A working model of the software development life cycle is presented, which describes the role and nature of software design within the context of software engineering. This model sets the stage for the more detailed sections on particular methods and techniques.

1.1 Scope

Software engineering literally encompasses all activities associated with producing software. The vastness of the topic and the fact that complex problems are being addressed by the practitioners of this art form have stifled attempts at describing its boundaries. But for the purposes of this treatment, the subject will include the spectrum of activities from analysis of requirements to installation. Although our emphasis will be on activities related to design, the role that design plays in the overall effort can be understood best if its relationship to software engineering is recognized.

1.2 A modeling approach

Subject areas as diverse as software engineering have been organized in different ways. For example, what mechanical engineering was thousands of years ago has evolved into a collection of specialty areas such as civil engineering, hydraulics, and heat transfer. Software engineering has had less time to mature. Hence, a natural structure has not evolved. Even worse, the complexity of problems being addressed is continually increasing, and consequently the state of the art is not yet well defined.

We could organize software engineering in various ways. For instance, we could identify a specialization of skills for each phase of software development. Another alternative would be to describe the functions performed by software engineers. We could also describe their activities, the tools they use, and what they build. But do these represent the basic or inherent properties that characterize the field?

One subject area that has had to deal with a similar sort of problem is biology, which makes sense out of (organizes) the immense volume of knowledge about living things. Take the case of plants: The great number of their characteristics — such as life span, method of reproduction, leaf size, shape, and preferred growing conditions — could give the person responsible for classifying them a nervous twitch. Fortunately, biologists have successfully developed a scheme with which to organize plants and other living things. Not a formal (mathematical) scheme, the approach is called morphological analysis [2].

Morphological analysis is extremely flexible in that it allows many different organizational schemes to be tried until one that provides some worthwhile insight or discovery is found. For example, the search for the so-called missing link was prompted, in part, by the use of an organizational scheme (or morphology) that characterized the developmental stages of man. The morphology revealed that the changes between stages seemed inordinately radical between two stages in particular. Hence, it was hypothesized that there may well have been another, as yet undiscovered, stage between