Software Maintenance Guidebook

Robert L. Glass
Ronald A. Noiseux

SOFTWARE MAINTENANCE GUIDEBOOK

Robert L. Glass
Ronald A. Noiseux

PRENTICE-HALL, INC.
Englewood Cliffs, New Jersey 07643

Library of Congress Cataloging in Publication Data

GLASS, ROBERT L date
Software maintenance guidebook.

Bibliography: p.
Includes index.
1. Electronic digital computers—Programming.
I. Noiseux, Ronald A., joint author. II. Title.
QA76.6.G56 001.64'2 80-21967
ISBN: 0-13-821728-9

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Editorial Production/Supervision by Theodore Pastrick
Manufacturing buyer: Joyce Levatino

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA PTY. LIMITED, *Sydney*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA PTE. LTD., *Singapore*
WHITEHALL BOOKS LIMITED, *Wellington, New Zealand*

SOFTWARE MAINTENANCE GUIDEBOOK

Robert L. Glass books published by Prentice-Hall

SOFTWARE RELIABILITY GUIDEBOOK

PREFACE

Maintenance is the enigma of software:
Enormous amounts of dollars are spent on it.
Little research or management attention is given to it.
And, in fact, it is not even a well-defined concept!

The time has come to begin removing the shroud of Merlinism which surrounds maintenance. This book is an attempt to do just that.

It begins by placing maintenance into the perspective of the "software life cycle" concept, and presents a definition of the term "maintenance."

It moves then to an unusual point of view — the importance of people in software maintenance. The theme is developed that the maintainer is an unsung hero, quietly keeping the computer software products humming in a world where little attention and few accolades are handed out.

With this people-oriented foundation, the book then moves to the meat of the subject — what technologies, both old and new, can be used by these unsung heroes? Tools and techniques which the maintainer should know about are described, ranging from the mundane (code reformatters) to the blue sky (the supercompiler). Heavy emphasis is put on the up-front activity of doing it right the first time — methods by which the software developer can ease the

problems of the software maintainer. A liberal number of examples is presented, most coded in the newly-emerging Department of Defense programming language Ada.

As a non-identical twin to the technologist point of view, the book then provides a management perspective on maintenance. Planning, organizing, and directing maintenance are all discussed. A somewhat radical view of software documentation is presented — one which has enormous promise for improved documentation quality, but one which requires new management thinking.

And finally, to bring the subject into focus and add a touch of reality, a maintainer's diary is presented — a day-to-day documenting of some of the events which characterize the software maintainer's on-the-job lifestyle.

Sprinkled throughout is a nearly complete bibliography of references to software maintenance in the literature. Symptomatic of the general disinterest in the subject, little has been written . . . until now.

The reader of this book is expected to be a software manager or technologist or student who has a basic understanding of what software is, but whose knowledge of maintenance is either rudimentary or has not been updated to include recent developments. It should be particularly useful to the consultant who wants to help computing organizations to a higher quality and more cost-effective maintenance activity; as a component in a university-level course in software engineering; and as on-the-job retraining material for experienced software people.

This book is written with an "equal opportunity" spirit! Neither racial nor sexual stereotypes should be inferred from words like "policeman" or "chairman," and the ubiquitous pronoun "he" should be taken as a third person substitute for the sexless version our language continues to lack.

ACKNOWLEDGEMENT

To the many of you who have contributed to this book, by virtue of an idea, a phrase, an article, or a book, our sincere thanks. The software world will become a better place because each of us strives to improve it.

DEDICATION

The authors dedicate this book to all the software maintainers of the world — those unsung heroes who quietly keep the software machinery humming.

CONTENTS

One

Introduction

Unfortunately, the nature of hardware and software errors differs in at least one fundamental characteristic—hardware deteriorates because of lack of maintenance, whereas software deteriorates because of the presence of maintenance.*

This is a pop quiz. Quick now, answer these questions without giving them deep thought. Just a simple yes or no will do.

1. Software maintenance consists of correcting the errors in software.
2. Software maintenance is an afterthought kind of thing, and little or no planning is needed for it.
3. Software maintenance consumes only a small slice of the software budget.
4. Software maintenance is a fairly uninteresting subject.

That's it. Just a simple introductory quiz. Now let's do a little self-grading.

* "Initial Thoughts on the Pebbleman Process," Institute for Defense Analyses, January 3, 1979; Fisher and Standish.

The correct answer to each of the questions above is "no." Starting with 100, subtract 25 points for each incorrect answer. If your grade is 100, you probably don't need to read much further in this book, except perhaps section 3, where some technology concepts are discussed. If your grade is 75, you are really on top of software as a profession, and deserve congratulations.

But if your grade is 50 or below, do not be chagrined. You are solidly in the majority of the software population. There are a lot of misconceptions and intuitional errors floating around about software maintenance, and you have fallen into them.

The reasons the answers to all of those questions are "no" will all be elaborated in this book. To satisfy your curiosity, though, in brief, those reasons are:

1. Software maintenance involves considerably more change implementation than it does error correction.
2. Traditional software maintenance approaches have pretty much been afterthoughts. However, the result of that inattention is frequently chaotic, unresponsive, and destructive maintenance.
3. Several studies show that software maintenance consumes over half of the software development dollar!
4. With that much money at stake, software maintenance just *has* to be interesting!

The purpose of this pop quiz is to sensitize the reader. Like other areas subject to a lot of stereotypes, software maintenance has been subjected to a veritable cloud of misunderstanding. Most experienced computing professionals and academicians, in spite of their broad knowledge of other aspects of software, are extremely naive about maintenance. It is the purpose of this book to dispel that naivete and those stereotypes, move past them, and provide some techniques and directions for the hopefully newly-interested-in-software-maintenance reader.

(Question 4, whether software maintenance is interesting, was in fact a trick question. Most honest software folk would have answered that it is, indeed, uninteresting. It will be a test of the effectiveness of this book to see if the reader can truthfully change that predicted

answer by the time he or she has finished this material!)

Now that you have had the stereotypes exposed, it is time for another pop quiz. Same rules, just a simple yes or no. But keep your guard up!

1. Because of its importance, software maintenance has been the subject of a lot of research studies.
2. Tools and techniques for the maintainer are well known.
3. Software maintenance people are usually the best the computing installation can afford.
4. Management places a lot of emphasis on software maintenance concerns.

Bet you were a little more wary this time! Question 3 was probably a giveaway. Once again the correct answer to each of those questions is "no." The fact of the matter, in a figurative nutshell, is that software maintenance has been a ho-hum subject to just about everybody. Researchers haven't bothered with it; tools and techniques tend to be leftovers from the software developer's toolbox; maintenance assignments are generally thought of as the pits and avoided by all but those who can't pull it off, like the junior folk and the lowest-rated ones; and management, in general, appears to be satisfied with that picture. (Some companies even have to give bonuses to induce people to do maintenance work!)

Like the previous one, this pop quiz had a purpose. If the first was designed to remove the stereotypes, this one was designed to show the aftereffects of those stereotypes. The fact of the matter is that software maintenance has been the subject of colossal blind neglect. Because its significance has been poorly understood, no one has cared to do much about it. The result is an understudied, extremely important technical field.

In that environment, it is difficult to write an effective and useful book. First, there is the problem of motivating readers into moving past the cover. As we have seen, "software maintenance" is a historic turnoff. To make matters worse, there is not a solid body of literature—the work of past maintenance experts—to base a book on.

The only reasonable answer to that dilemma, at least at present, is to create a pioneering book and make it interesting enough to attract

readers who are otherwise turned away by the topic. As mentioned before, you readers get to issue the grades on *that* pop quiz.

The authors of this book have spent a lot of time maintaining software. They are also students of computer science, from both an experiential and an academic point of view. This book is an attempt to transfer that experience in a meaningful way; to provide a basic reference point from which the desperately needed future research into software maintenance can proceed; to give the practicing software professional and his or her manager the essential information needed to perform a vital function more effectively; and to provide the student of software engineering with both perspective and tools to fully understand the whole spectrum of software activity. Even including that historically uninteresting field, software maintenance!

1.1 MAINTENANCE IN THE LIFE CYCLE

At a recent computing conference, discussion of the so-called computing life cycle became a standing joke. Every presenter of every paper showed a viewfoil or a slide containing his or her graphic version of the concept. Toward the end of the day, one wag referred to his as the "obligatory software-life-cycle chart"!

The field of software engineering is by no means immune from fads. A subject catches the eye of the researcher, and 90% of researchers end up pouring some energy into that subject. A buzz word comes along with a new or interesting connotation, and the field is alive with that buzz word. A concept emerges that promises lowered production costs and better schedule performance in the delivery of software, and everyone leaps aboard the bandwagon.

Fortunately, most of these fads have value. Perhaps not as much value as the number of bandwagon jumpers would indicate, but still value. The software life cycle is just such a concept, with just such a value.

The major thrust of this concept is to dispel the myth that the process of software development is principally the act of coding software. Some early studies [1] in the late 1960s and early 1970s began to cast doubts upon that myth and to focus attention on what really did constitute the software development process. Further study showed that the analysis of software requirements and the design of the

software consumed far more time than its coding. Just as important, further studies also showed that the checkout and testing of the coded software consumed an even more surprisingly large amount of time. The emergence of the software-life-cycle chart was an attempt to reflect these findings and to graphically illustrate the situation that the factual studies were just beginning to expose.

Many early software-life-cycle charts left out entirely the subject of software maintenance. Just as the early focus had been mistakenly placed on the coding process, the newer focus was still mistakenly placed on the development process. It was not until the mid-1970s that software-life-cycle charts began including software maintenance—and what a profound awakening those pieces of graphic art produced!

It is time for a definition. The *software life cycle* will be defined here as the entire process, from beginning to end, of the development and use of software. That process includes the distinct phases called requirements definition, design, coding, checkout and testing, and maintenance. The reason for calling it a life cycle is probably clear from its definition—it is a womb-to-tomb type of definition, one that pays attention to all of the activities in the software process.

The reason this concept emerged and became popular is only partly dependent on the myths that it exposed and overturned. It is also popular because, in the world where software costs are a major concern, it was becoming obvious that spending money on one software-life-cycle phase could have a profound effect on some of the other phases. For example, putting money (and presumably care) into design would lower the cost of coding, checkout, and testing. Better coding would reduce checkout costs. More effective requirements analysis would lower the cost of every subsequent phase.

This cost-trade argument really became important when the significance of maintenance became clearer. Doing a better job in any of the preceding phases always had a cumulative effect on lowering maintenance costs. Considering how large maintenance costs are, this is of pretty fundamental importance. One whole section of this book, and a lot of comments along the way, will be focused on the task of lowering maintenance costs by spending more careful effort on the up-front life-cycle phases (section 3.2.2.1).

Just as important, the cost trades, prior to life-cycle consideration, had not been obvious. The software producer who delivered software to a customer but had no responsibility to maintain it, for example,

might do a slovenly job on some aspects of software development in order to cut costs, with the customer actually incurring higher total cost because of the increased cost of maintaining a slovenly product. With the attention on life-cycle considerations, the customer could direct the developer to do a better job, pay a little more for that better job, and end up with a product whose total cost was less. These economic arguments were fundamental to the popularity of the life-cycle concept.

It is time for a few more definitions. The various phases of the software life cycle are made more specific here.

The first phase in this book will be called *requirements/specifications*. Elsewhere it may be called systems analysis. It is the phase where the problem is being understood and defined. A solution to the problem may evolve during the requirements/specification phase, but it is held in check pending a complete understanding of the problem. Only then should a solution be consciously considered; its representation is then stated in terms of a specification for a software system, and that specification is the primary output of the requirements/specification phase. Perhaps the greatest hazard during this phase is the temptation to define a solution to part of the problem, ignoring the hard parts or those that are ill-defined. Succumbing to this temptation leads to inadequate design and implementation, which in turn leads to revised requirements and major modifications (or, in fact, to "death" of the system). Modification of existing software is probably the greatest plague of the software profession. It is difficult, costly, and frustrating. Many a program has been thrown away and rewritten because it was "unmodifiable." Thus, well-thought-through requirements and specifications are vital to the eventual activities of the software maintainer.

The second phase is the *design* phase. It is time to translate the problem and its requirements specification into a conceptual solution, a blueprint for the actual solution or implementation that will follow. Computing-specific considerations are made: What computer? Which of its resources, and how much? What language? What modules? What sequence of functions? What data structures? What else? All of these ingredients are dumped into the specifications-defined pot, and stirred into a workable and specific plan. The primary output of the design phase is a design representation. It may take the form of words, flowcharts, decision tables, program design language, or any number