STANDARDIZED DEVELOPMENT OF COMPUTER SOFTWARE PART 1 METHODS

Robert C. Tausworthe

STANDARDIZED DEVELOPMENT OF COMPUTER SOFTWARE

Robert C. Tausworthe

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

PART 1
METHODS

PRENTICE-HALL, Inc., Englewood Cliffs, New Jersey 07632

Published in 1977 by Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632

Printed in the United States of America

10 9 8 7 6 5 4 3

ISBN: 0-13-842195-1

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA PTY. LIMITED, *Sydney*
PRENTICE-HALL OF CANADA, PTD., *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*

STANDARDIZED DEVELOPMENT OF COMPUTER SOFTWARE

PREFACE

This monograph started as a set of rules given piecemeal as standards to a team developing a conversational, incrementally-compiled, machine-independent version of the Dartmouth BASIC language for the Jet Propulsion Laboratory, called MBASIC. (Originally, "M" stood for "management-oriented", indicating its intended set of users; however, its great flexibility and ease of use has since won over many scientific and engineering users as well.) The first draft was a mere collection of the first sketchy set of rules, along with some background material on structured programming.

As the design progressed, the emphasis expanded from the design of a language processor to a project developing software methodology using the MBASIC development as a testbed activity. New rules were supplied as necessary and old ones had to be revised or discarded. Some of the ones that sounded so good when first imposed had effects just opposite to what was desired. The MBASIC design and documentation standards underwent several complete iterations, each under new rules to calibrate their effectiveness. The working drafts of this monograph were in constant revision to maintain a current set of standards for the project.

Further expansions of the working drafts were made to include much tutorial material, since I used portions of the text as lecture topics for graduate-level computer science classes at West Coast University and for seminars in software standards at the Jet Propulsion Laboratory. Interactions with the students and professional programmers with widely different backgrounds proved to be very enlightening.

I realize that some who have already "learned programming" may find fault with what they read here. I hope their objections are mostly with how the rules impact their personal programming style. Style is a reflection of a programmer's personal programming *habits* and his own *preferences* in

the way he does things. If the rules given here don't work, that is another matter.

What I have attempted to do is to merge individual disciplines and good practices into a methodology that neither destroys personal style nor reduces motivation and involvement. The given set of rules is the base for a consistent and effective methodology; but there may be other equally effective and consistent methodologies. I do not allege to profess the *only* way toward improved software development—just one that works.

The monograph does not reflect, nor is it meant to reflect, exact standards or practices now in effect at JPL; however, much of the material has formed the basis for Deep Space Network software guidelines and standard practices currently in effect.

Several individuals at the Jet Propulsion Laboratory have reviewed the drafts and many have provided rules, suggestions, and other material. I have expected such criticism, and welcomed constructive material by any who cared to supply it. I have tried to be open to all correct, potentially worthwhile ways to improve the development of software and to build these into a uniform coordinated methodology for programming, a set of rules universally sound.

I offer one apology at the outset—for my literary style. About half-way through writing this monograph, I was suddenly surprised to learn that I often referred to software development personnel in the masculine. Lest I be accused of male chauvanism, let me attempt to defend myself by explaining that the references appear thus because I tended to place myself in the roles of these individuals. In writing, I also tended to be addressing myself, rather than any envisioned reader or actual software development person. By the time I realized I might be taken to task for this by distaff readers, the style was set and writing was too far along—another case where a software error was discovered too late to change the product without having major schedule and economic impact!

I would like particularly to acknowledge the aid given to me in the form of encouragement, ideas, criticisms, reviews, questions, and informative discussions by Walter K. Victor, Mahlon Easterling, Robert Holzman, James Layland, Robert Chamberlain, Edward Posner, Daniel Preska, Richard Morris, and Henry Kleine of the Jet Propulsion Laboratory, and Daniel Lewis, Frank Bracher, John MacMillan, Richard Jaffee, and Howard Mayberry of National Information Systems, Inc. Also, I want to express my appreciation to Margaret Seymour for typing the many drafts and to Shozo Murakami for editorial assistance on this final version.

Finally, I wish to thank those who have attended the many seminars and classes given from this work during its various stages of completion; many insights into the secrets of software engineering across a broad programmer base occurred to me as the result of these classroom discussions.

Robert C. Tausworthe

STANDARDIZED DEVELOPMENT OF COMPUTER SOFTWARE

CONTENTS

I. INTRODUCTION

A computer system is a rigid, dispassionate machine; it is designed and built to react in definite, microscopically precise ways to programmed commands. The program it executes comprises a large collection of atomic instructions organized into macroscopic algorithms and computational procedures in performance of a desired task. The differences between a hoped-for behavior and the actual are evidences of *human failures* to instruct the computer properly. Nevertheless, such failures are referred to as "errors in the program" or "bugs", and justly so—the servant has executed but cannot comprehend any reasoning behind the instructions given it. Moreover, it has constrained the human capacity to communicate in doing even this much, as it has required instructions in its own programming language, rather than in more human terms.

Computer programs have thus, from the very first been subject to error—missteps in coding committed by the programmer—and then not discovered until after the program's operation can be examined and seen to be in error. The cause of such errors may then be either obvious, very elusive, or somewhere in between. In any case, the diagnosis comes *after the fact*, as the computer proceeds at such a pace as to make concurrent diagnoses out of the question. Once diagnosed, any subsequent (trial) corrections must be

rerun to validate the proper response, at extra expense. The human proclivity to err in programming is probably the singularly most prominent, overriding factor against producing economical, reliable software.

Because the computer lacks judgement itself and responds to direction totally ignorant of the task to be done, programmers attempt to build in some measure of quasi-judgement by instructing the device to perform certain tests on input and to check for known or probable processing anomalies. They may instruct the computer, based on such information, to take some less abrasive action than complete failure. Such programming practices are often called "user forgiving", "error insensitivity", or "defensive". Regardless of the terminology, such practices are attempts to establish the proper master/servant relationship, whereby the machine adapts to the human, rather than vice-versa.

R. Holzman, a colleague at JPL, once remarked (1972) "When you can tell a computer, 'Oh, *you* know what I mean!'—and it *does*—then that's a computer language!" The industry, of course, may never attain that goal of man/machine communication, but it is reaching. In its reaching, it has made several significant progressions to define methods, procedures, and standards for use by programmers to reduce the number and severity of their "program errors".

Among the first significant developments were the inventions of higher-level languages, language processors, and the provisions for programmers to annotate their programs with some form of rationale for their own benefit. In addition, novice programmers learned to draw flowcharts, as a prelude to coding, as a means of developing their skill, and as a method for designing the program procedure—the algorithm scoping the task. But programmers still made errors, at about the same rate per instruction as they had previously. The only difference was that as many errors did not reach the run-time stage, and each instruction did more in a higher-level language. Still more higher-level languages have been developed; until today, there are probably as many programming languages as there are natural languages.

At some point, programmers, or their supervisors, or their customers, recognized that, even though a program might be working, no one could understand *how* it was working well enough to make changes without introducing a lot of side-effect errors, or how *well* it was working enough to assess the programming quality. So the idea, "document what you have coded so I can understand it", sprang up. Managerial seminars developed methods to cajole and coerce [1,2] designers, programmers, coders, *et al.*, to document. The necessity to document [3] was evident to all who had to

read and maintain the software, but dreaded by the documentor. Flowcharting was a nuisance and rarely matched the code, regardless which was produced first. Annotations of the code were in a similar state, as were narrative descriptions. Since the computer cannot execute a flowchart, narrative, or annotation anyway (only the code), and the human was just as likely to err in describing his code as he was in coding it, other systems emerged: self-documenting code, automatic flowcharting, standardized documentation formats, etc. Computer technology was beginning to evolve into an engineering discipline.

1.1 THE NEED FOR SOFTWARE STANDARDS

Years ago, the cost of computing was largely in machine costs; now the larger portion is paid to people developing, using, and maintaining programs. In fact, the trend in computing costs is the complete dominance of manpower costs over machine costs.

Software is big business; the indirect costs caused by failures to meet schedule or performance requirements often exceed the costs of the software itself, because software development always seems to be on the "critical path" of a system development. Boehm [4] suggested the following prescription for software headaches:

a. Get software off the critical path in system development.

b. Increase software productivity.

c. Improve software management.

d. Get an earlier start.

e. Make software responsive to actual user needs.

f. Increase software reliability.

The present monograph is an attempt to provide *formal disciplines* for increasing the probability of securing software that is characterized by high degrees of initial correctness, readability, and maintainability, and to promote *practices* that aid in the consistent and orderly development of a total software system within schedule and budgetary constraints. These disciplines and practices are set forth as a set of rules to be applied during software development to eliminate (this is the goal)—or at least to drastically reduce—the time spent debugging the code, to increase understandability among those who come in contact with it—especially managers, who must often make decisions relative to competing resources (such as budget, schedule, execution speed, memory size, etc.)—and to

facilitate operation and alteration of the program as the requirements or program environment evolves.

To be effective, I recognize that a set of standards must not be *imposed* so much as *adopted.* But once a set is adopted, its rules should be enforced. Needless to say, some of the rules I give are broad and, therefore, open to interpretation. I have tried to make these as specific as I could without destroying their general applicability. But some vagueness may yet remain.

One may question whether the strict adherence to definition, design, production, testing, and documentation rules hamper programmer creativity or decrease his motivation and involvement; this has not, in my experience, turned out to be the case. Programming methodology tends to be rather scantily taught in computer-science courses in the universities. What methodology a programmer possesses he may have had to learn largely for himself, tutored by his own coding, discovered osmotically from reading programs others have written, or found through discussion with his peers. Programmers, as any problem-solvers, generally *welcome* a workable, well-disciplined approach to problem solving, so they do not have to re-invent the wheel, so they know what is expected of them and how they will be judged on their performance, so they know what level of reporting is required, and so they can really get into the design and make a clean, good, well-operating system.

Good standards enforce themselves. Once the programmer recognizes that his own performance is improved by standardized methods, he is its foremost proponent. When he suddenly realizes that he is capable of understanding a program written by someone else, he is convinced forever. I have personally seen instances where experienced programmers have at first rebelled at the entire concept, but once forced, they recognized the benefits derived, assisted in further development, and helped enforce standards in their own organizations.

The reports from industry are equally encouraging. Although productivity indices tend to be highly variable across wide ranges of applications and across software development personnel, nevertheless, analysis of quantitative data [5] indicates that the standards forming the basis of this monograph generally produce better than 50% improvement in overall project productivity. This overall productivity figure includes analysis, design, testing, management, support, and documentation, in addition to coding and debugging. Moreover, the figures in support of this improvement have been computed in terms of delivered code—the incidental effort spent in developing code used to support the production and code, which later had to be replaced, have not been counted.

1.2 SOFTWARE DEVELOPMENT

At the outset of a programming project, there are only a problem (program requirement) and a programming language in which the solution to that problem is to be stated. In between, there is the gap to be bridged by the development process.

The actual creative process which goes on in a program designer's mind is certainly not well understood. It probably rambles from broad concept to details and completeness, and, perhaps on occasion, from detail to the broader concept.

When writing a paper or preparing a talk, one first jots down notes, then an outline of the material to be covered. After the outline is expanded by way of a few iterations, the narrative is written. Many revisions are usually necessary if the paper or speech is to be of any significance.

A piece of software probably should not be much different in the way it is created. Successive refinements and revisions of a program are going to be necessary if it is to be of high quality.

Moreover, the revision process in software development is unavoidable. People cannot think of everything, in the right order, correctly, in one pass (Figure 1-1). One can hope, however, that there are procedures that tend to let the creative process take a natural course, but yet minimize the probability that, at some advanced stage of development, one must "throw out the whole thing and start all over from scratch."

One of the most costly ways to develop software is to begin the production phase before the program definition and design have reached an adequate state of completion. A small change in the program definition, for example, can avalanche down through the work done, resulting in suboptimal design, patched programs and code, introduction of undesirable side effects, and excessive debugging time.

The pressure of a schedule and the awareness that a great deal of coding has to be done cause many managers to let the design or coding begin, anyway, just to get started on a job that is obviously huge. Hence, the process of design is begun throughout the system at the very bottom before the design has been properly thought out and precisely defined at the top. A classical "bottom-up" design emerges, leading to difficulty in integrating the resulting components in a system.

Yet cooperative interaction between the definition, design, and production activities associated with developing a program can be mutually beneficial when properly interfaced. The proper interface in this

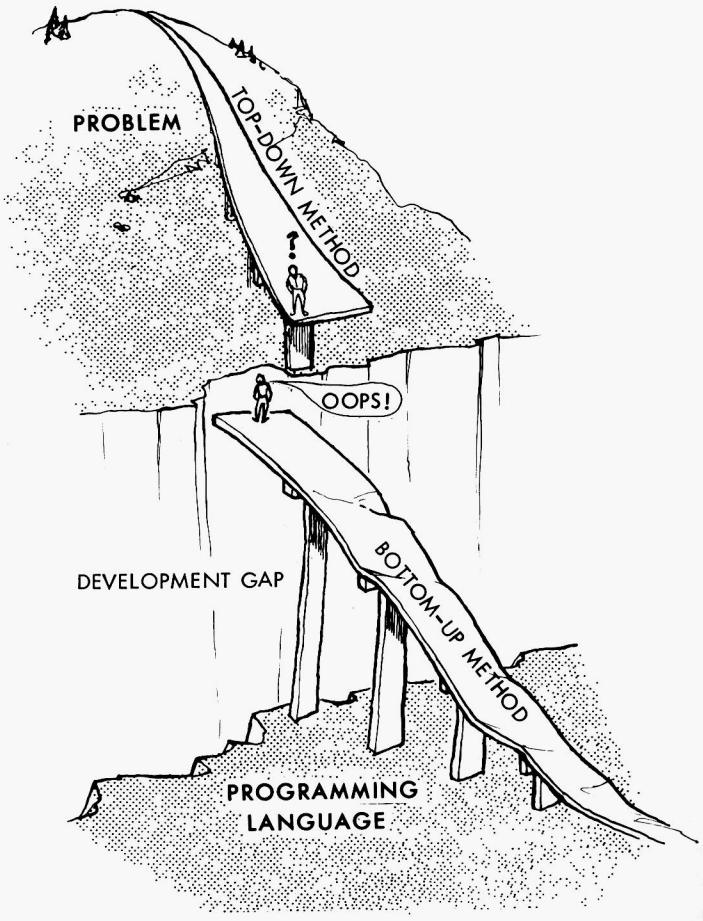

Figure 1-1. Bridging the software gap

context is an organization of the tasks to permit revisions and refinements without requiring extensive rework.

The procedures of this monograph have evolved from the belief that successive refinement of a concept by *adding* more and more detail is a less costly, more certain discipline than refinement by succesive *alterations* of the original concept.

One principle by which program concepts evolve in a natural, structured way emerged from Dijkstra's work in the THE Multiprogramming System [6]. He conceived that a program could be organized into hierarchic levels of support. The principle, known as *levels of abstraction* (see Sec. 2.5), formed the basis for what has become known since as *structured*