Software Engineering Environments

CONCEPTS AND TECHNOLOGY

Robert N. Charette

Software Engineering Environments

CONCEPTS AND TECHNOLOGY

Robert N. Charette

Intertext Publications, Inc.
McGraw-Hill, Inc. New York, NY

Dedication

To Mo.

Library of Congress Catalog Card Number 86-81063

10 9 8 7 6 5 4 3 2 1

ISBN 0-07-010645-2

Intertext Publications, Inc.
McGraw-Hill Book Company
1221 Avenue of the Americas
New York, NY, 10020

ACKNOWLEDGEMENTS

It is a very difficult thing to write out the acknowledgement section of this book because so many people have been involved in its making. One always risks leaving someone out. Therefore, I want to thank first all those unnamed, but important people who have contributed ideas and have argued with me over the last few years to help clarify my thoughts on the subject. Individual members of the SEEWG, of NSIA, of NASA, of STARS, and SofTech, Inc. have provided the most help. It really has been you people that have shaped and developed the core contributions of this book, and deserve the majority of the credit.

I want to thank Paul Fortier of the Naval Underwater Systems Center for giving me the opportunity to write this book, and for commenting on and giving advice on how to improve many sections of it. I want to thank also Tom Conrad of the Naval Underwater Systems Center and Bob Converse of Computer Sciences Corporation for dedicating a tremendous amount of their precious free time also reviewing some of the initial drafts, and for setting me on the straight and narrow whenever I would craft some dubious idea.

I especially wish to thank Dr. Charles McKay of the University of Houston at Clear Lake City. Dr. McKay has been very instrumental in forcing a rethinking of many of my thoughts about environments, and for making me understand the problems of building software engineering environments for extremely large software systems like NASA's Space Station.

Special thanks go out to Commander Kate Paige, USN and Hank Stuebing of the Naval Air Development Center who have been the most influencial people in my thinking about environments. It would be difficult to find two people that have contributed more to the present direction of software engineering environments than Kate and Hank.

Finally, I want to thank a very special friend who kept me going throughout the agony of writing this book. She was the one who patiently listened to my complaints, encouraged me when I felt overwhelmed, and kicked me when I needed it. Books don't get written without those people around. To Maureen Albrecht of SofTech, my greatest thanks. Thanks, Mo.

Robert Charette
1 June 1986

TABLE OF CONTENTS

1. SOFTWARE ENGINEERING

"Software is like entropy. It is difficult to grasp, weighs nothing, and obeys the Second Law of Thermodynamics; i.e., it always increases."

Augustine's Laws[1]

1.0 Introduction

Imagine, for a few minutes, that the time is the early 1990s. Your job, if you decide to accept it, is to determine the software support requirements of a new, computer-intensive system that a company you are interviewing with has just won a competition to build. This means you will be responsible for specifying all the computer-related elements required by the engineers, scientists, technicians, managers, etc. involved in the project. They will use these elements to create, maintain, and manage the application software that is going to be developed and used within the new system. To help you understand the nature of the job, a brief description of the system envisioned has been provided.

First, the proposed system architecture will contain several dozen heterogeneous computers networked together, but dispersed geographically across thousands of miles. Moreover, the system will possess a large, as yet undetermined, number of space-based elements continuously communicating with the network via ground and satellite links.

The system's reliability requirements are severe. No individual failure of a computer-based component of the system can keep a software task in execution from completion, nor can two such failures result in damage to life or property. Furthermore, when any software updates are inserted into the system, the system must continue to execute; i.e., the system cannot be brought down during system regenerations. Once the system is made operational, it must continue to be operational throughout its lifetime.

Since the system will likely evolve over the next twenty years or more, easy technology insertion for both hardware and software is mandatory. First estimates are that somewhere around 100 million new lines of high-level code will be necessary for the system to fulfill its currently prescribed mission, and up to 300 million additional needed to support currently foreseen missions.

The system must be built within seven years, on time, within a tight budget under public and congressional scrutiny, and any failure of the system will be described in excrutiating detail by the world's press. And it better be fully documented. And, oh yes, there will be over 4,000 personnel, in thirty major companies, and countless minor ones, involved in the total software development. These companies will be located across the United States, and possibly Europe and Japan, and must be able to "share" the support system you are specifying.

Although grossly simplified, the system described above represents some of the requirements that will have to be met if NASA's Space Station Project is to meet its initial operational capability in the mid-1990s. Ambitious? Most definitely, but it is not alone in sheer size, cost, complexity, schedule, or necessity. For instance, WIS and the Strategic Defense Initiative (SDI) also fall into this category of "ambitious" systems, as do many others currently on the planning boards of both the commercial and government sector.2

Although the systems mentioned above are on the larger side of the size and complexity spectrum, many computer systems currently in everyday use are of similar scope. The financial transaction systems used at banks, the billing systems used by telephone companies, typical company payroll systems, and the Social Security and Veteran's Administration check distribution systems are only a few examples that twenty-five years ago would have been considered as ambitious as the Space Station, WIS, and SDI systems are today. The airline reservation system is a prime example of a system most of us take for granted, but which, on reflection, is rather impressive. Travel agents from anywhere in the United States can call into a computerized network to buy a ticket on almost any airline on one of thousands of specific flights to hundreds of domestic and international cities, select your seat (smoking, non-smoking, window, or aisle), order special assistance such as meals or wheelchairs, if necessary, and get your boarding pass — all within a few minutes without your ever having to set foot inside an airline terminal. And if it takes more than a few minutes, we get annoyed!

Both national security and domestic tranquility have become highly dependent on our computerized systems. However, increasingly there is something disquieting about this same level of dependence. Airplane crashes (e.g., the 1979 Air New Zealand crash in Antartica), aborted space launches (e.g., the Space Shuttle on its first two launch attempts in 1981), ghost trains (e.g., the San Francisco Muni Metro train that "wasn't there" in 1983), and runaway missiles (e.g., the Soviet cruise missile that "got away" over Norway and Finland in 1985) are only a few examples of what can happen when a computer system doesn't work correctly. Less spectacular events may result only in the nuisance of lost time or money as when your automatic bank teller swallows your bankcard because it wasn't reprogrammed to accept old bankcards when the new ones were issued. On the other hand, computer system failure can be potentially catastrophic to hundreds of millions of people. That's because now even our concept of national defense is predicated on what is achievable through the capabilities provided by computers. Some don't believe that is necessarily good.

For instance, Dr. David L. Parnas, recognized as one of the United States' leading computer scientists, does not believe that systems like the SDI, a cornerstone in our defense policy for the twenty-first century, can be built with the reliability necessary to accomplish its objective of protecting the United States from nuclear attack. This, he believes, is in large part because we neither understand enough about the system requirements nor the process of software creation to implement it [PARNAS85]. If this view is correct, the result is wasted money at best and, as others argue, possibly reduced national security.

Whether or not these specific views about SDI are totally valid — and many believe they are not — Parnas does point out some concerns that few will argue with. It seems most software development efforts are plagued by the fact that we don't know exactly what we are building, nor how to build it successfully. More specifically, if new systems such as the Space Station are ever going to be successfully realized, a number of serious problems that confront the current state of the practice of building computerized systems must be overcome. We can summarize the current situation with three points:

- *The High Cost of Software*–Software costs, in both financial and human terms, are increasing rapidly as computerization of society spreads. Software in large systems may approach $4 billion in initial costs alone, while a computer error in an accounting system may mean immeasurable anguish to an elderly person denied a much needed check. Software costs are the dominant costs of software-based systems today.

- *The Variation in Practice*—There is a wide range of software practices exercised both within and across government and commercial sectors. These practices may lag the state of the art by as many as fifteen years. As a result, software is difficult to manage and varies widely in cost, reliability, and maintainability.

- *The Need for Increased Productivity*—Based upon analysis of projected software requirements over the next two decades, demand will outstrip the resources for the production of software. The only viable answer is to increase productivity, which can be accomplished on the scale required only by increasing the productivity of the process as a whole. Failure to increase productivity tends to increase the severity of the two problems above.

Attacking these problems and creating successful software systems is the theme of this book. But before we explore this theme in detail, it would be beneficial to review a condensed history of how these software problems came to be. To paraphrase Santayana, if we don't understand our past, we are condemned to repeat it.

1.1 Setting the Stage: "The Software Problem"

In the early days of digital computing, software development was aimed primarily at getting a single specialized program to work. Computers, then as now, were meant to save large amounts of manual labor. First generation computers and programs in the 1940s were mainly targeted to the compilation of the massive astronomic navigation and ballistic tables used primarily by scientists and the military. In the 1950s, as second generation computers provided more power and flexibility, and made it cost effective to use computers, the trend shifted away from the purely scientific and military usage of computers to the processing of business data.

In both the 1940s and 50s, a program was considered increasingly successful if it: a) executed; b) executed quickly; c) gave an acceptable answer. Regardless of the success criteria the quality of the program was highly dependent on the skill of the programmer. Because of the lack of resources provided by the computer to work with, there was great admiration for an individual programmer's ingenuity at providing the maximum computation using the least amount of resources.[3]

New memory technology spawned the third generation computers. These computers allowed even larger programs to be executed, and many programs to be executed concurrently. This additional capability moved the trend in software development in the late 1950s and early 1960s away from single programs or sets of small programs to large assemblages of programs linked to do one integrated system function. Large-scale use of software was probably first attempted in the SAGE (Semi-Automatic Ground Environment) air defense system. It contained a computer with 58,000 vacuum tubes, consumed 1.5 megawatts of power, executed a real-time application program of 100,000 instructions, and had a support system of 112 million instructions [AUGUSTINE83].

The creators of the SAGE system were the unfortunate first, but by no means the last, to experience the problems surrounding the development of large-scale software systems. Most of the problems they faced are still familiar to those creating software systems today.

For example, the application program was too large and complex to be created by a small group of programmers, instead requiring large teams of programmers. The increased number of personnel, along with their required logistical support (notice the large support system — much larger in fact than the system being built), rapidly boosted the cost of the project. The problem of scale also appeared. A large program just didn't seem to work all that well [YEH84]. It became clear that as the program size increased, the probability for error increased even faster. Although reliable statistics aren't available, SAGE probably also used most of the programmers then in existance, thus depleting the numbers available for other programming jobs in the commercial sector and driving up labor costs. Finally, a larger program and a more capable computer allowed the programmers to be even more "ingenious" than before.

However, the problems and lessons learned in the encounter with the SAGE system development were to be quickly dismissed as a period of great fervor and optimism swept the computer field in the mid and late 1960s. Newer, more capable computers and software techniques promised to make any lessons learned obsolete or marginally transferable to new system developments.

It was widely believed — or at least the marketing people claimed — the advent of the fourth-generation machines would allow the distribution of programs across networks to form systems magnitudes larger than those ever envisioned by the SAGE designers. Moreover, the costs of the hardware systems were constantly going to be reduced, and what were previously scarce resources (processing speed, memory, etc.) were going to become everyday items. Additionally, to handle the software development of these large systems, it was recognized that some discipline would have to be brought to the software field. But here, too, progress seemed to be making rapid headway.

After all, research efforts were reporting results in the areas of software design disciplines, problem abstraction, and notations for software representations. These factors would allow a rational approach to software development. Simultaneously, high-level programming languages first developed in the 1950s were finally gaining acceptance by the programming community as the code generated by the compilers became ever more reliable and efficient. The issues of programmer productivity and software quality, although still not completely solved, were seen as things of the past. The age of functional programming as seen in the 1940s,'50s, and early'60s would be replaced by the new age of structured programming [PETERS80]. The problems brought out by the development of the SAGE system would soon be solved.

However, by the late 1960s, this confidence was beginning to wane as more and more software system developments were encountering exactly the same problems as the SAGE system. So in 1968 and 1969, at two exciting and controversial NATO conferences held in West Germany, new ways were discussed to solve the "software problem," typified by expensive, unreliable, and unmaintainable software. A new term was coined to help express what was thought to be the solution: Software Engineering.

SOFTWARE ENGINEERING. The beginnings of software engineering, given some literary license, could make a great mythological tale in the tradition of Wagner. It began with the creation of a brother to hardware engineering, whose highly developed discipline was able to create systems that, unlike software, worked within design constraints. Envious, a rebel group of computer scientists sought to combine into a disciplined engineering approach the magical techniques used by the self-styled programming artisans who then controlled the creation of software. In 1969, these disciples of the engineering approach met in Europe and created the term "software engineering," which was to change forever the way of developing software. Software engineering was to mean "the establishment and use of sound engineering principles in order to economically obtain, software that is reliable and works efficiently on real machines" [NAUER69]. However, many opposed this view, calling it a fabrication that took away the uniqueness and magic of programming. So the great debates over whether the creation of software was an art, science, or discipline ravaged the land for almost twenty years [HOARE84].

Over the last few years the debates have died down as those who believe in the engineering approach seem to have won the battle for converts. Some cynical observers blame it on better public relations, although others say that since software still doesn't work well, neither those favoring the science or art approaches want to take the blame. Nevertheless, the term "software engineering" has received a certain level of acceptance throughout the software community and is the current thrust. However, exactly what it means is still somewhat fuzzy, and this is probably why everyone accepts it. It seems to mean whatever one wants it to mean.

This shouldn't be surprising to anyone either, as software engineering hasn't yet reached its twentieth birthday, and like most teenagers, still can't decide on what it wants to be. The IEEE Standard Glossary on Software Engineering Terminology [IEEE83] defines it as, "the systematic approach to the development, operation, maintenance, and retirement of software." Software is then defined as "computer programs, procedures, rules and possibly associated documentation and data pertaining to the operation of a computer system."4

In Fairley's text [FAIRLEY84] on software engineering concepts, software engineering is defined as "the technological and managerial discipline concerned with systematic production and maintenance of software products that are developed on time and within cost estimates." A software product includes the system-level software, application software, and all associated documentation. Some typical items making up a software product are shown in table 1-1. Rather than adding a new definition to the list, we will be content to use Fairley's definition as our operational one. However, we do wish to add one more element to the discussion.

The above definition of software engineering describes what it is, but only implicitly its goals. Pressman [PRESSMAN82] states them very succinctly: "The key objectives of software engineering are (to define, create, and apply5) (1) a well-defined methodology that addresses a software life

• Requirements Document	• Specification Document
• System Description Document	• Test Specification
• Software Development Plan	• Test Procedures
• Functional Design Document	• Test Reports
• Detailed Design Document	• User Manual
• Verification Plan	• Source Code Listing
• Trouble Reports	• Object Code Listings

Table 1 - 1 **Typical Components of a Software Product**

cycle of planning, development, and maintenance; (2) an established set of software components that documents each step in the life cycle and shows traceability from step to step, and (3) a set of predictable milestones that can be reviewed at regular intervals throughout the software life cycle." The combination of both these definitions allows the derivation of some of the goals/objectives of the software engineering process, which are shown in table 1-2.

So with the combination of these two thoughts still fresh in our minds, let's look at software engineering as it appears today.

1.2 Software Engineering Today

As has been amply documented elsewhere [BOEHM76, WEGNER78, REDWINE84] advances in software engineering technology have occurred on all fronts: requirements analysis, implementation strategies, cost models, etc., to name just a few. Each advance has been aimed at reducing one of the three problems we mentioned earlier: (1) the high cost of software, (2) the variation in practice, or (3) the lack of productivity. In reality, each problem is coupled tightly to the other, and solutions to one usually help to make the solutions to the others more feasible. For instance, the movement toward

Goals/Objectives of the Process of Software Engineering

- To improve the accuracy, performance, and efficiency of the overall product under development
- To apply well-defined methodologies for the resolution of software/system issues
- To provide rational resolution of conflicts and documentation of differences when resolution is not possible
- To provide for product change in response to new or modified requirements
- To provide an understanding of the role of all stakeholders in the resolution of complex issues and the differing sets of constraints under which they operate
- To provide clear communication among management and the members of the system/software engineering teams
- To provide the understanding of how current systems and the evolution of future products are impacted by present day decisions
- To provide explicit identification and consideration of all normally implicit tradeoffs, assumptions, constraints, and intentions, and recognition of what is, and is not, important for planning and decision making purposes
- To provide anticipation of contingencies and identification of the impacts of proposed situations
- To document decisions, the rationale behind decisions, and the actions taken; create and maintain a corporate memory
- To provide explicit descriptions of schedules and milestones, and an understanding of the effects of time upon issues under consideration

Table 1 - 2

using high-level languages in software development instead of assembly language can be traced primarily to improving programmer productivity. Although taken for granted now, even just ten years ago this was a major change of operating practice in some industry organizations. This single change has accounted for billions of dollars saved in software costs, the

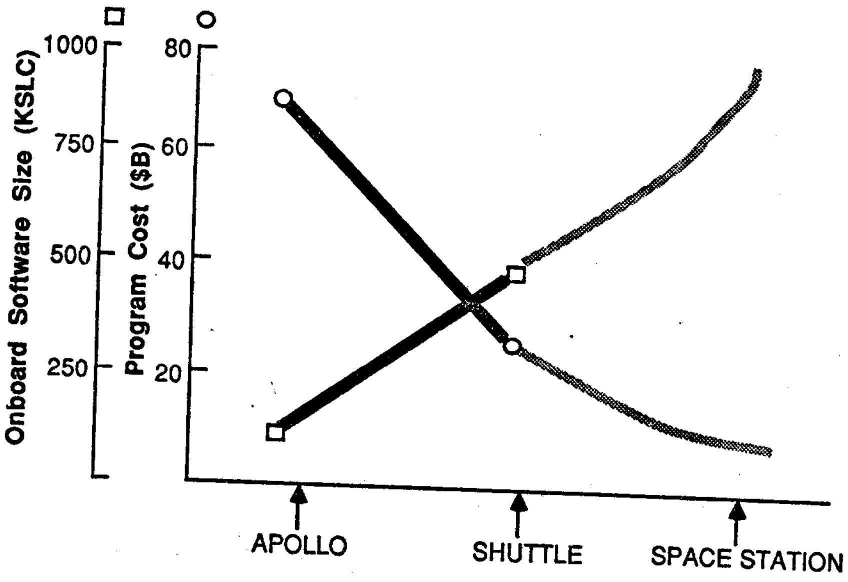

Figure 1 -1 **Software Cost ratio vs. Space Flight Software Trends**

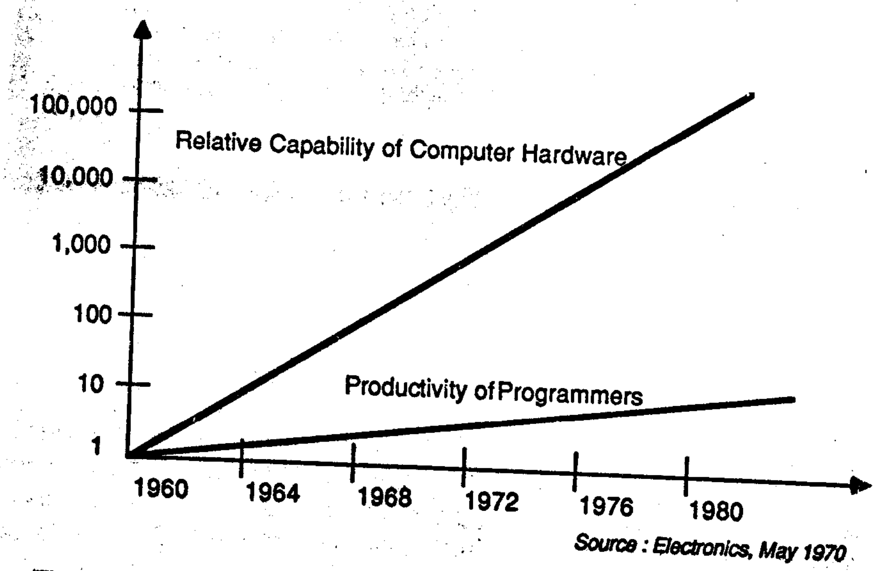

Figure 1 - 2 Relative Capability of Computer Hardware over Time