Software Reliability

PRINCIPLES & PRACTICES

Glenford J. Myers

[illegible]
M 996

Software Reliability

Principles and Practices

Glenford J. Myers

Staff Member,
IBM Systems Research Institute
Lecturer in Computer Science,
Polytechnic Institute of New York

A WILEY-INTERSCIENCE PUBLICATION
JOHN WILEY & SONS
New York • London • Sydney • Toronto

[illegible]

Library of Congress Cataloging in Publication Data:

Myers, Glenford J 1946-
Software reliability.

(Business data processing, a Wiley series)
"A Wiley-Interscience publication."
Bibliography: p.
1. Computer programs—Reliability. I. Title.

QA76.6.M93 001.6'425 76-22202
ISBN 0-471-62765-8

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Preface

In the late 1960s, the fact that computer systems can make mistakes, and that these mistakes can have an effect on our lives, received a considerable amount of attention in the press. A typical "COMPUTER GOOFS!" news story might have described a department store customer who received a bill for $0.00, the customer's attempts to communicate this error to the store, and an endless stream of notices from the computer threatening to terminate the customer's account unless the bill was paid. Often, the easiest solution was actually to mail a check to the store for $0.00 or, as in one case, to have the incident publicized in the financial section of the *New York Times*.

Problems like this one are certainly undesirable, but their harm is usually limited to personal inconvenience. I would rather have an argument with a department store computer than be caught in a runaway computer-controlled train where a programming error is causing the train to attempt to accelerate to 1000 kilometers an hour instead of 100 kilometers an hour. In today's world errors such as this are now possible. Because data processing is touching our lives to an increasing extent, computer errors can now have such consequences as property damage, invasion of privacy, personal injury, and even loss of life.

This book was written to describe solutions to the problem of unreliable software. Every aspect of software production is examined, and solutions are expressed in two forms: *principles* and *practices*. Principles are major strategical approaches to the production of reliable software. Practices are smaller-scale tactical solutions to various aspects of the unreliability problem.

The book consists of four major parts. The first part defines software reliability, discusses the principal causes of software errors, and attempts to motivate the reader to read the remainder of the book. Part 2 contains a large set of principles and practices used in the design of reliable software. I use the word *design* in a broad sense, starting with the definition of requirements for a software system and ending with the coding of individual program statements.

Part 3 covers the broad area of software testing, a set of processes that consume vast percentages of data processing budgets and yet about which very little is known by most people. Although the key contributor to reliability is precise design, testing plays an important role in software reliability. A large set of proven principles and practices for testing is discussed.

Computer professionals will recognize that there are additional variables that affect reliability; many of these are discussed in Part 4. For instance, organizational structures, personalities, attitudes, management plans and motivation, programming tools, and the work environment all have a significant bearing on reliability. Part 4 also describes problems in current programming languages and computer architectures and suggests solutions to these problems. The subjects of mathematically proving program correctness and predictive techniques (reliability models) are also discussed.

This book only treats software reliability from the point of view of software errors. A large topic, the use of software to correct or circumvent hardware failures such as input/output device failures, is not covered in this book. Although this topic must obviously be considered in total system reliability, it is excluded because it is a separable topic that deserves the attention of another book and, in comparison to software reliability, it is fairly well understood.

This book should benefit anyone having an interest in the production of reliable software. People directly involved in software production, such as programmers, analysts, test personnel, programming managers, and data processing managers, should benefit the most. Programming language designers should find the material on programming languages, programming style, and computer architecture of interest. Software users, particularly those responsible for purchasing software or writing contracts for the development of new software, will gain insight into reliability and how it may affect their use of computer systems. Researchers should find all the material useful in obtaining an understanding of the reliability problem, examining the usefulness of past research, and understanding the promising areas for future research.

The book should be useful both as a reference book and as a text. The book will expose the university student to many of the real-world problems of software development. As a textbook, it can be used in senior or graduate-level computer science or software engineering courses on software development, providing that it is supplemented with actual experience such as a class project. I have used all of this material in graduate-level courses on software reliability at the IBM Systems Research Institute and the Polytechnic Institute of New York.

I thank many of my collegues at the IBM Systems Research Institute for valuable suggestions on the book. In particular, R. Goldberg and C. H. Haspel provided constructive criticism of most of the book, and B. G. Weitzenhoffer, C. J. Bontempo, and J. E. Flanagan provided valuable advice on material in Part IV. I must point out that certain material in the book can be considered as controversial; the opinions and views expressed are solely my own and do not necessarily represent the opinions and views of the people mentioned above nor the IBM Corporation.

GLENFORD J. MYERS

New York, New York
April 1976

Contents

PART 1

Concepts of Software Reliability

CHAPTER 1

Definition of Software Reliability

The most significant problem facing the data processing business today is the software problem that is manifested in two major complaints: software is too expensive and software is unreliable. Most computer professionals recognize the former problem as largely a symptom of the latter. Because of the unreliable nature of today's software, considerable expense is incurred in software testing and servicing. Although this book focuses on the problem of unreliable software, the problem of high cost is indirectly confronted because of its relationship to unreliability.

It is interesting to note that the software reliability problem as it exists today was observed in the early days of computing:

> Those who regularly code for fast electronic computers will have learned from bitter experience that a large fraction of the time spent in preparing calculations for the machine is taken up in removing the blunders that have been made in drawing up the programme. With the aid of common sense and checking subroutines the majority of mistakes are quickly found and rectified. Some errors, however, are sufficiently obscure to escape detection for a surprisingly long time [1].

This observation was published by three British mathematicians in 1952. Although software errors were encountered before 1952, this seems to be the first recognition of the reliability problem, that is, that a considerable amount of time is required for testing and, even after this, some software errors still remain undetected.

IS THE MOON AN ENEMY ROCKET?

The immediate problem encountered in dealing with software reliability is one of definition: What is a software error and what is software reliability? Agreeing on standard definitions is important for avoiding such problems as a system user's stating that an error exists in his system and the system developer's replying, "No, it was designed that way."

The Ballistic Missile Early Warning System is supposed to monitor objects moving toward the United States and, if the object is unidentified, to initiate a sequence of defensive procedures starting with attempts to establish communications with the object and progressing potentially through physical interception and retaliation. An early version of this system mistook the rising moon for a missile heading over the northern hemisphere. Is this an error? From the user's (Defense Department's) point of view, it is. From the system developer's point of view, it may not be. The system developer may take the position that the requirements or specifications stated that action should be initiated for any moving object appearing over the horizon that is not a known friendly aircraft.

The point here is that different people have different views as to what constitutes a software error. Before we can begin discussing methods to eliminate software errors, we have to establish a definition of software errors. Rather than developing a definition from scratch, we can benefit by analyzing common definitions of software errors and determining their weaknesses.

WHAT IS AN ERROR?

One common definition is that a software error occurs when the software does not perform according to its specifications. This definition has one fundamental flaw: it tacitly assumes that the specifications are correct. This is rarely, if ever, a valid assumption; one of the major sources of errors is the writing of specifications. If the software product does not perform according to its specifications, an error is probably present. However, if the product does perform according to its specifications, we cannot say that the product has no errors.

A second common definition is that an error occurs when the software does not perform according to its specifications providing that it is used within its design limits. This definition is actually poorer than the first one. If a system is accidentally used beyond its design limits, the system must exhibit some reasonable behavior. If not, it has an error. Consider an air-traffic-control system that is tracking and coordinating aircraft over a particular geographic sector. Suppose that the original contract for the system specified that the system should be able to simultaneously handle up to 200 aircraft. However, on one particular day and for some conceivable reason, 201 aircraft appear in this sector. If the software system performs unexpectedly, say it forgets about one plane or aborts, then the software contains an error, even though it is not being used within its design limits.

A third possible definition is that an error occurs when the software does not behave according to the official documentation or publications supplied to the user. Unfortunately, this definition also has several flaws. There exists the possibility that the software *does* behave according to the official publications but errors are present because both the software and the publications are in error. A second problem occurs because of the tendency of user publications to describe only the expected and planned use of the software. Suppose that we have a user manual for a time-sharing system that states, "To enter a new command press the attention key once and type the command." Suppose that a user presses the attention key twice by accident and the software system fails because its designers did not plan for this condition. The system obviously contains an error, but we cannot really state that the system is not behaving according to its publications.

The last definition that is sometimes used defines an error as a failure of the software to perform according to the original contract or document of user requirements. Although this definition is an improvement over the previous three, it also has several flaws. If the user requirements state that the system should have a software error mean-time-to-failure of 100 hours and the actual operational system proves to have a mean-time-to-failure of 150 hours, the system still has errors (because its mean-time-to-failure is finite), even though it exceeds the user requirements. Also, written user requirements are rarely detailed enough to describe the desired behavior of the software under all possible circumstances.

There is, however, a reasonable definition of a software error that solves the aforementioned problems:

A software error is present when the software does not do what the user reasonably expects it to do. A software failure is an occurrence of a software error.

I expect two reactions to this definition. The software user's reaction will be, "Precisely!" The software developer may react with, "The definition is impractical, for how can I possibly know what the user reasonably expects?" The point is that the software developer, to design a successful system, must always understand what the system users "reasonably expect."

The word "reasonably" was placed in the definition to exclude such situations as a person's walking up to an information retrieval terminal in a public library and asking it to determine how much money he has in his checking account at the local bank. The word "user" describes any human being that is entering input into the system, examining output, or interacting with the system in any other way. A large software system (application programs, operating system, compilers, utility programs, and so on) will have a large number of different users, such as people communicating with the software through remote terminals or the mails (often people with no knowledge of computers or programming), application programmers, system programmers, and system operators.

The reader should now be able to grasp an elusive characteristic of software reliability: *software errors are not an inherent property of software.* That is, no matter how long we stare at (or test, or "prove") a program (or a program and its specifications), we can never find all of its errors. We may find a few errors, such as an endless loop, but, because of the basic nature of software errors, we can never expect to find them all. In short, the presence of an error is a function of both the software and the expectations of its users.

Although I will use this definition as the basic definition of a software error, it does have at least one flaw. Consider an airlines reservation terminal that instructs the clerk to "ENTER FLIGHT NUMBER AND DATE" to which he responds by entering "239,MAY10." The system responds with the message "INCORRECT DATE" because it expected the date in the form 10MAY. Is this a software error? According to our definition it may be, but I say it is not; it is probably a human factors problem. We could broaden the definition to cover situations such as this one by viewing them as human factors