Software Requirements
Analysis and Specification

ALAN M. DAVIS

Software Requirements
Analysis and Specification

ALAN M. DAVIS
BTG, Inc.

Prentice Hall, Englev[illegible] Cliffs New Jersey 07632

Library of Congress Cataloging-in-Publication Data

Davis. Alan M. (Alan Michael)
Software requirements : analysis and specification / Alan M. Davis.
p. cm.
Includes bibliographical references
ISBN 0-13-824673-4
1. Computer software—Development. I. Title.
QA76.76D47D38 1990
005.1'2—dc20 89-16392
CIP

Editorial/production supervision: ***bookworks***
Cover design: ***Victoria A. Heim***
Manufacturing buyer: ***Robert Anderson***

About the cover: The selection of Victoria Heim's "Hands on Hands" was inspired by the *What* vs. *How Dilemma* described in section 1. 2. 1.

A Division of Simon & Schuster
Englewood Cliffs, New Jersey 07632

The publisher offers discounts on this book when ordered in bulk quantities. For more information write:

Special Sales/College Marketing
Prentice-Hall, Inc.
College Technical and Reference Division
Englewood Cliffs, New Jersey 07632

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-824673-4

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

List of Illustrations

Foreword

Over the past fifteen years, there has been a great deal of concern about the high cost of software. On one hand projections show that demand for applications outstrips our society's ability to produce them at this time. The software tools applied to assist users, including programmers, in developing solutions are improving only incrementally. On the other hand, the U.S. software industry is saddled with more than $300 billion worth of ill-structured and difficult-to-maintain software inventory. As a consequence the cost of maintenance in large data processing centers has exceeded 60% of their budgets. These problems call for improving and automating the software development process, and while much has been done, analysis and specification of requirements remain a relatively untouched area. Yet it is perhaps the most important aspect of any large software development project.

Beginning in the mid 1970s, there have been a number of techniques and systems developed for the purpose of analyzing and defining requirements. The leading methods include SADT, PSL/PSA, SREM, E-R Data Model, and Data Flow Diagram à la Yourdon. Numerous studies have been conducted to analyze and compare these methods, while others have concentrated on using the techniques/methods. Although proponents of each methodology suggest that their method can suitably carry out the entire analysis process, we have learned that each problem requires a different set of techniques and tools. Therefore it is very refreshing to read this book by Alan M. Davis. I was pleasantly surprised by his heavy emphasis on the fundamental issue of problem solving rather than on techniques or tools. The implicit process model

of interactive analysis and specification is very effective and realistic. I particularly appreciate the notion that a requirement specification binds only the solution space and sometimes it is necessary to do some design or even implementation (via prototyping) to determine where the solution boundary lies. This contrasts drastically to some of the pure requirements methodology of stating only *what* and not *how*. For as Davis clearly points out, the *how* of one level is the *what* of another level.

In addition to presenting software requirements analysis as a problem-solving activity, this book has three outstanding features. First of all this is one of the most readable technical books I have encountered. Secondly this book provides a broad coverage of various methodologies, languages, and tools. It also contains a thorough reference list that will benefit anyone interested in this topic. Finally the book presents numerous examples throughout to illustrate both problem-solving principles and techniques applied to requirements analysis.

I consider *Software Requirements: Analysis and Specification* to be a significant contribution to the software engineering field and would recommend its use in a university-level software engineering curriculum.

Raymond T. Yeh
Austin, Texas

Preface

This book focuses on the early phases of the software development life cycle. These early activities are commonly called *software requirements analysis* or *software requirements specification.* I have written this book for two audiences—(1) the practicing systems engineer, software analyst, and requirements writer; and (2) the advanced student of software engineering who wishes to receive specialized education in the early phases of the software development life cycle.

This book is unique because it discusses the latest research results from the requirements arena but at the same time is highly practical. Some authors on the subjects of requirements, specifications, or analysis stress a particular technique and then try to convince you to embrace that technique as *the* technique to apply to your requirements problem. Some authors present compilations of other authors' primary works, which in turn advocate particular techniques. I, however, will not try to convince you that any particular technique will always be right. Instead this book will arm you with a thorough understanding of (1) what you need to accomplish during the requirements phase, (2) how each of a wide variety of techniques can help you accomplish some part of that task, (3) how different aspects of your particular application will strongly suggest using one technique or another, (4) how to compare and contrast all techniques using some common terminology, and (5) how to find a technique that will assist you in analyzing your problem and specifying your product's requirements instead of one that provides you with yet another problem (that is, figuring out how to use the technique itself). I believe that a

good technique should lend itself to your problem—not insist that you mold your problem to fit the technique, and this philosophy permeates this book. A good friend of mine, the late Professor Donald B. Gillies, once said, "If you program in Algol long enough, you start to see the entire universe as an Algol program" [GIL72]. Of course this is true not only of Algol and many other programming languages but many engineering techniques, including those used during requirements.

I often hear analysts asking such questions as, "Should I use Structured Analysis or SREM?", or "Should I use USE.IT or SADT?" It is true that all four of these are termed "requirements" techniques by their respective inventors (and all are discussed in this book). However the posers of these questions demonstrate an inherent lack of understanding of the requirements domain and the same naivete as someone who asks, "Should I wear my black shoes or my leather gloves today?" There is no trade-off involved; Structured Analysis and SREM serve two completely different purposes, and USE.IT and SADT serve two completely different purposes. This book provides a new taxonomy of requirements-related activities to enable you to ask the right questions and provide sensible answers to them as well.

After reading this book, you may expect to be able to do the following:

- Given any real-world problem, to organize your ideas to quickly find loose ends that require further analysis and areas that have been overanalyzed.

- Given any real-world problem, to select a set of requirements techniques, tools, and/or languages that will aid in analyzing that problem.

- Once you thoroughly understand your problem, to formulate and organize a specification of the solution system's required external behavior completely, consistently, and unambiguously.

- To select a set of requirements techniques, tools, and/or languages that could be used to augment your specification of external behavior to help alleviate inconsistencies, incompletenesses, and ambiguities.

- Given a document defining software requirements for a system, to determine where it is overspecified, underspecified, inconsistent, ambiguous, or incomplete.[1]

- When presented with "yet another (new) requirements technique," to determine (1) how it relates to other techniques and (2) whether it is applicable to your individual problem.

[1]It is interesting to note that there are no hard and fast rules for this determination. As you will see in reading this book, the correct level of these attributes varies dramatically with the stage of development. For example, a document whose purpose is to define needs and invite potential developers to bid competitively to satisfy those needs must be much more open ended than one whose purpose is to define the to-be-built system's external behavior just prior to software design.

Techniques that are presented in this book are followed by case studies showing how the technique can be applied to aspects of three real problems. The same three problems are used as case studies throughout the book to help you compare and contrast the techniques. The three problems were deliberately selected to represent three very different application domains:

Problem	Application Attributes
1. Automation of a book distribution company	Data intensive; some aspects highly human interactive; other aspects highly batch; multiple simultaneous actions
2. Automation of helicopter landing	Hardware control intensive; synchrony intensive; time sensitive; nondeterministic
3. Transportation of people from New York to Tokyo in 30 minutes	A *very* difficult problem[2]

Clearly no problem is entirely batch, entirely difficult, or entirely data intensive. Every problem has a bit of each of these attributes. The important thing to remember is to employ a technique that makes the difficult parts easier. Therefore when faced with a particular problem, first determine what the most difficult parts are, then find the problem in the preceding list that is most similar to yours, and employ techniques most suitable to that problem.

This book is organized into seven chapters plus an extensive annotated bibliography.

Chapter 1, the Introduction, sets the stage by (1) describing where the software industry is today, (2) motivating the tremendous need for improved software engineering techniques, (3) showing where requirements analysis and specification fit into the total software development life cycle, (4) defining precisely what requirements are (and are not), (5) explaining fundamental differences between problem analysis, and product description, and (6) providing conclusive evidence that failure to detect requirements defects is a major cause of skyrocketing software costs. The chapter concludes with a thorough discussion of software applications in general and the three case studies used throughout the book.

In Chapter 1 we learned that there are two fundamentally different things being done during the requirements phase—problem analysis and product description. Chapter 2 explores the former topic in depth, and Chapters 3–5 explore the latter. The bulk of the second chapter describes, compares, contrasts, and

[2]Selecting a very difficult problem as an example in this book has its advantages and its disadvantages. The primary advantage is to help the reader understand how to approach such a problem. The greatest disadvantage is that if this is truly a difficult problem, we will not solve it and in fact should make little headway in solving it (or it would have already been partially solved). Unfortunately this lack of progress may lead some readers to believe that the techniques employed are not useful. The correct conclusion is that solving a really difficult problem is not easy: You simply chip away at small pieces, brainstorm a lot, and hopefully solve it. As Turski [TUR80] said, "To every hard problem, there is a simple solution, and it's wrong."

applies a variety of problem analysis techniques. However prior to that discussion, fundamental principles underlying problem analysis techniques are described. The chapter concludes with examples of applying each of the techniques from the chapter to the three case studies described at the end of Chapter 1.

Chapter 3 introduces the subject of how to write or evaluate a document (that is, the software requirements specifications—SRS) that specifies the external behavior of a software product. A list is provided of all attributes that a "perfect" SRS should exhibit (realizing of course that no SRS can ever be perfect!). Each of these attributes is defined, and many examples from actual SRSs are given to demonstrate each attribute. The chapter concludes with sample outlines for SRSs that can be used as checklists for the novice SRS writer.

In Chapter 3 we learned that there are two types of requirements that belong in an SRS—behavioral and nonbehavioral. Chapter 4 explores the former category of requirements (Chapter 5 explores the latter). The bulk of this chapter describes, compares, contrasts, and applies a variety of techniques that can be used to describe the external behavior of software. Like Chapter 2, Chapter 4 concludes with examples of applying each of the SRS techniques described in this chapter to the same three case studies.

In addition to describing external functional behavior, a properly written SRS also describes the "ilities" of the software. Namely, it describes how adaptable, how maintainable, how reliable, etc. the software should be. Chapter 5 defines many of the attributes of a software product that must be addressed in the SRS to ensure that the as-built product satisfies real needs. Guidance and examples are provided to help you (1) decide which "ilities" should be emphasized in your particular application and (2) see how to specify the product traits in as unambiguous a manner as possible.

Prototyping has been used in engineering disciplines for years but only recently received attention in software engineering. There are two schools of prototyping during the requirements phase—throwaway and evolutionary. Proponents of both schools call what they do simply "prototyping;" rarely is a distinction made in practice. Unfortunately, if you build either type of prototype expecting to achieve results available from the other, you will be grossly disappointed. Chapter 6 thoroughly describes the preceding two types of requirements and explains their respective impact on the requirements process, the software development life cycle, productivity, and product success.

Chapter 7 summarizes key ideas presented in Chapters 1–6, explains where the requirements field is going, and where it is likely to be in the next fifteen to twenty years.

The Glossary defines terms used with special meanings in the requirements domain.

The annotated Bibliography offers a compilation of approximately 600 published articles, books, and reports on the subject of requirements. Many of them are described in a short synopsis.

Depending on how you wish to use the book, you may want to read it in a number of different ways:

If you are a software practitioner who wants to learn about requirements and the types of techniques, tools, and languages available, I suggest you read the entire book. If you have a particular problem and do not know where to turn for advice on how to analyze it, I suggest you read Chapters 1, 2, and 6 only. If you have been asked to write an SRS for a product to be built by your own organization, you should read Chapters 1, 3–6.

If you have been asked to review an existing or proposed SRS, Chapters 1, 3, 4, and 5 can help you.

If you are using this book as a reference to help you find an applicable technique, tool, or language, read Sections 1.1 and 1.2 to gain an appreciation for the difference between problem analysis and writing an SRS—then browse through Chapters 2 and 4 to find appropriate approaches.

Notes to the Teacher

If you are using this book as a text for a graduate or an advanced undergraduate-level course, let me indicate how I organize the course when I teach it:

Topic	Textbook References	Activity	Hours[a]
Administrative introduction	N/A	Lecture/discussion	1.2
Introduction			
Software life cycle	1.1	Lecture/discussion	0.7
What are requirements?	1.2	Lecture/discussion	3.3
Exercise 1 (SRS evaluation)	4	Student team exercise	2.8
Introduction (continued)			
Why are requirements so important?	1.3	Lecture/discussion	0.8
Taxonomy of applications	1.4 & 1.5	Lecture/discussion	0.5
Problem analysis	2	Lecture/discussion	8.0
Exercise 2 (problem analysis)	2	Group participation	4.0
The SRS	3	Lecture/discussion	2.3
Specifying behavioral requirements	4	Lecture/discussion	4.0
Exercise 3 (SRS evaluation)	4	Student team exercise	4.0
Specifying nonbehavioral requirements	5	Lecture/discussion	4.0
Requirements prototyping	6	Lecture/discussion	1.2
Summary	7	Lecture/discussion	1.0
Exam review	N/A	Discussion	2.5
Exam	N/A	Exam	2.7
		Total hours	43.0

[a]Actual class contact time.

Exercises 1 and 3 represent before-the-course and after-the-course exercises on how to recognize inadequacies in an SRS. In both cases, I distribute copies of actual SRSs, then divide the class into teams of three to five each to assess independently the quality of the SRS. Usually I also give each team a unique role to play:

1. Design team
   Wants to be able to build software based on SRS
   Disdains overspecification
2. System testing team
   Wants to be able to test that the software product meets its requirements
   Can not tolerate ambiguity
3. System user/customer team
   Wants to be sure product is worth paying for
   Wants an understandable document
   Can not tolerate underspecification
4. Requirements Consultants, Inc., team
   Wants to see formality
   Intolerant of ambiguity

During Exercise 3, I usually walk from team to team to assist each in playing its role realistically. This is followed by formal 15-minute presentations by each team to the entire class. Because each team has a unique position, much controversy is generated concerning the appropriateness of the SRS. In both exercises, students learn how to recognize inadequacies in an SRS. In Exercise 3, they also learn that there are no clear-cut answers to the question, "What is a perfect SRS?"

In Exercise 2, the assembled class simulates the brainstorming that goes on during a typical problem analysis session. I serve as moderator and provide little added value other than as a poser of key questions when the students lose momentum. In this way students learn how to use problem analysis techniques to organize ideas.

REFERENCES

[GIL72] Gillies, D.B. Private communication. Lawrence, Kans., January 1972.

[TUR80] Turski, V. Stated orally at IFIPS Congress 1980, Tokyo. October 1980.

ACKNOWLEDGMENTS

Only one author's name appears on the cover of this book, but I did not write it alone. Dozens of collegues, friends, and relatives provided assistance of all kinds, and without that assistance, this book would never have existed.

Dr. Edward Bersoff, president of BTG, Inc., deserves the most thanks. His moral and financial support, technical ideas, friendship, and high standards of integrity have been inspirational to me and crucial to the creation of this book.

During the ten-year period when the ideas expressed in this book were developed, I had the opportunity of discussing requirements-related issues with many people. Among them are a few individuals who stand out for having provided me with considerable insight into the vast challenges associated with analyzing problems and writing software requirements specifications: Dr. Ed. Bersoff of BTG, Inc.; Peter Coad of Object International, Inc.; Ed Comer of Software Productivity Solutions, Inc.; Dr. B. Dasarathy of Concurrent Computers; Bruce Gregor of RS Data Systems Inc. the Software Productivity Consortium; Tom Miller, formerly with GTE Laboratories; Dr. Tom Rauscher of Xerox Corporation; and Dr. Ray Yeh of International Systems Corporation. Dr. Jim Sherrill and the U.S. Army systems automation officers in the "Defining Software Requirements" course at the Computer Science School in Ft. Benjamin Harrison, Indiana, from March 1986 to December 1987 also deserve special recognition for having unknowingly served as a test bed for many of the ideas in this book. The feedback that they supplied about my successes and failures with teaching this material was used to fine tune the instructional methods used in that course as well as in this book. Early reviewers of this book, namely, Dr. Bob Glass of the *Journal of Systems and Software* and Bill Cureton of Sun Microsystems, Inc., provided much help with the rough spots in early manuscripts. Bruce Gregor was invaluable in developing the Pfleeger Pfliers case study.

A number of people played a key role in creating this book, although they did not realize it at the time. Three individuals in my early professional life gave me inspiration, taught me the meaning of self-confidence, and shaped the very nature of how I think today. These people, Dick Dworak, the late Dr. Don Gillies, and Dr. Tom Wilcox, probably have more to do with who I am today professionally than any others.

The Prentice Hall editor Paul Becker was helpful in providing sound advice about writing this book. He gave me just the right number of reminders to make sure I was always progressing and on the right track. In addition to being a trusted friend, Ms. Marilynn Bersoff deserves my praise and thanks for maintaining her incredible level of quality standards in the physical production of the text and figures that comprise this work. Ms. Eileen Bates and Interactive Development Environments, Inc., provided access to Software