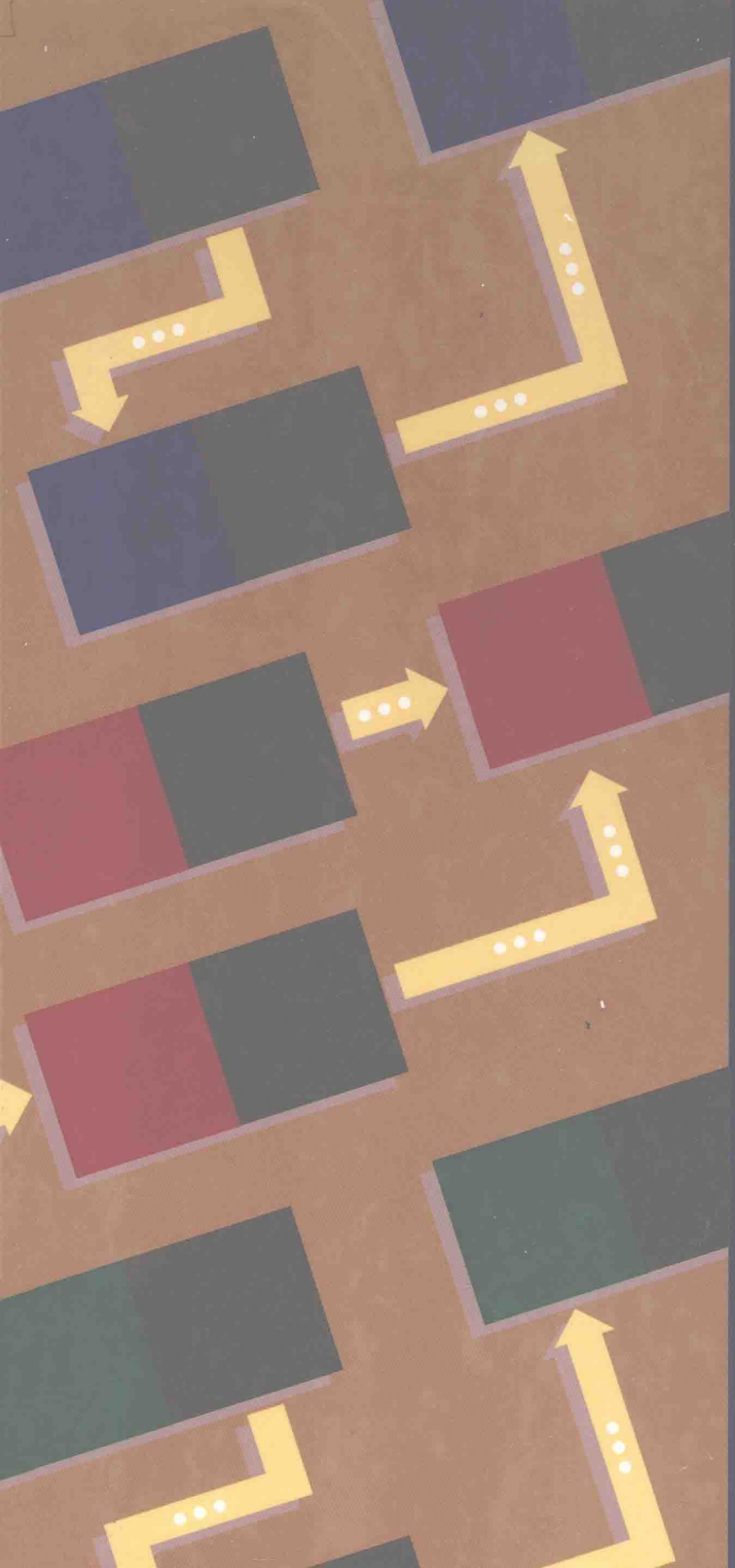


CONCEPTS IN DATA STRUCTURES & SOFTWARE DEVELOPMENT

A TEXT FOR THE SECOND COURSE IN COMPUTER SCIENCE

G. MICHAEL SCHNEIDER
STEVEN C. BRUELL

CONCEPTS IN DATA STRUCTURES AND SOFTWARE DEVELOPMENT

A Text for the Second Course in Computer Science

G. Michael Schneider
Macalester College

Steven C. Bruell
University of Iowa

WEST PUBLISHING COMPANY
St. Paul New York Los Angeles San Francisco

COPYEDITING: Pam McMurry
TEXT DESIGN: Lucy Lesiak Design
TECHNICAL ILLUSTRATIONS: G&S Typesetters, Inc.
COMPOSITION: G&S Typesetters, Inc.
COVER DESIGN: David J. Farr, Imagesmythe, Inc.

610 Opperman Drive
P.O. Box 64526
St. Paul, MN 55164-1003

Printed in the United States of America

98 97 96 8 7 6 5 4 3 2

Library of Congress Cataloging-in-Publication Data

Schneider, G. Michael.
Concepts in data structures and software development : a text for the second course in computer science / G. Michael Schneider, Steven C. Bruell.
p. cm.
Includes index.
ISBN 0-314-77460-2 (hard)
1. Data structures (Computer science) 2. Computer software—Development.
I. Bruell, Steven C. II. Title.
QA76.9.D35S36 1991
005.1—dc20

Printed with Printwise
Environmentally Advanced Water Washable Ink

90-42783
CIP

CONCEPTS IN DATA STRUCTURES AND SOFTWARE DEVELOPMENT

A Text for the Second Course in Computer Science

I dedicate this book to my wife Ruthann, my children Benjamin and Rebecca, and my sister Karen.

G. Michael Schneider

To Sandy

Steven C. Bruell

ABOUT THE AUTHORS

G. Michael Schneider

G. Michael Schneider is currently Professor of computer science and Director of the computer science program at Macalester College in St. Paul, Minnesota, where he has been since 1982. Prior to that he was on the faculty of computer science at the University of Minnesota for eight years.

In addition to his research interests, he is the author of a number of best-selling college textbooks, including *Introduction to Programming and Problem Solving with Pascal, Advanced Programming and Problem Solving with Pascal* (with Steven C. Bruell), and *Principles of Computer Organization*. For the past ten years he has been extremely active in computer science education and curriculum development through membership in the ACM Special Interest Group in Computer Science Education (SIGCSE), the national Computer Science Accreditation Board (CSAB), and the Consortium of Liberal Arts Computer Science Programs (LACS). He has written and presented papers on a range of issues related to the undergraduate curriculum and the design and use of computer laboratories. He is the recipient of two recent National Science Foundation grants to create formal teaching and research laboratories and to study how they can be integrated into the undergraduate computer science curriculum.

Professor Schneider received his Ph.D. in computer science from the University of Wisconsin with a research specialization in computer networks, distributed systems, and parallel processing.

Steven C. Bruell

Steven C. Bruell received the B.A. degree in mathematics (with honors) from the University of Texas at Austin in 1973, and the M.S. and Ph.D. degrees in computer science from Purdue University, West Lafayette, Indiana, in 1975 and 1978, respectively.

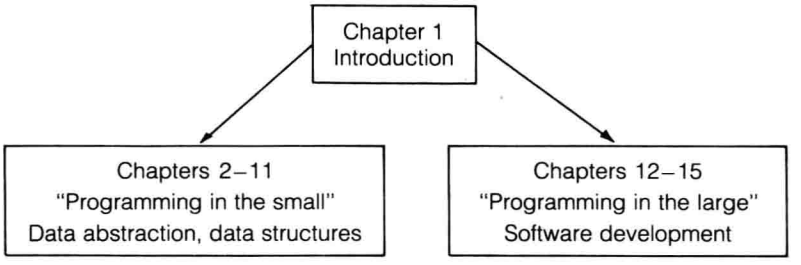
He is currently an Associate Professor of computer science at the University of Iowa, Iowa City. His current research interests include distributed systems, parallel simulation, and computer performance evaluation. He has coauthored many papers and three books. He has been an ACM National Lecturer and is currently serving as the Associate Chair of the Computer Science Department.

PREFACE

This book is intended for the second course in computer science, the course called CS 2 by the ACM in its curriculum recommendations. It assumes that the reader has completed a traditional first course that introduced programming in a high-level language, most likely Pascal. In this text we investigate the topics of data structures and software development.

The material is divided into two parts. Part I (chapters 2 through 11) covers abstract data types, data structures, recursion, and the analysis of algorithms. This section can be viewed as a study of "programming in the small" because it treats issues related to the construction of correct, efficient, and maintainable program units. Part II (chapters 12 through 15) addresses topics related to the software life cycle—specification, design, implementation, validation, and maintenance. This part can be viewed as a study of "programming in the large," since it looks at issues related to the construction of software systems made up of hundreds or thousands of individual units of the type discussed in Part I.

Pictorially, the organization of this book is:



The placement of the discussion on software development following the data abstraction and data structures chapters does not imply that the material must be presented in that order. We have tried to keep these two sections independent from a teaching perspective, and the instructor who wishes to reverse the order of presentation and treat the material on software development first can do so without difficulty.

The material in Part I of this text is a continuation of ideas begun in CS 1. In that first course, students focused on the construction of correct, robust, and reliable program units through a study of algorithm design, structured programming, data types, elementary data structures, procedural abstraction, empirical testing, and programming style. In Part I we continue that investigation and present concepts that permit the design of program units with increased robustness, efficiency, shareability, and maintainability. The topics covered include the specification, design, and implementation of abstract data types (chapter 2) and advanced data structures such as stacks, queues, strings (chapters 3 and 4), trees (chapters 7 and 8), sets (chapter 9), and graphs (chapter 10). The topics of sorting and searching are woven throughout the chapters of Part I, but in chapter 11, we present a separate discussion on the important topic of external sorting and searching. In chapter 5, we introduce recursion so that the student can understand and utilize recursively defined data structures and feel comfortable using recursion as a general problem-solving paradigm. Finally, in chapter 6, we provide the mathematical tools needed for analyzing the time and space efficiency of algorithms, including sequential, recursive, and parallel algorithms.

When presenting the topic of data structures in Part I, we have been guided by and strongly influenced by the following four principles:

1. A student should first study a data structure in terms of its high-level behavior before becoming enmeshed in the details of its implementation. It is not that these implementation details are unimportant. However, the separation of the external and internal properties of a data structure allows us to look first at the formal properties of a data type, such as the nature of operations on objects of that type. Only after these fundamental properties are fully understood do we move on and look at how this structure can be implemented in a typical high-level language. The tool that allows us to create this conceptual separation is the abstract data type (abbreviated ADT), and it is the central theme of Part I. It is the method used to present the data structures studied in chapters 2 through 11. We first study a data structure as an abstract data type, concentrating only on its formal properties and its operations. Later we look at ways in which this structure can be implemented, and we investigate efficiency issues related to the choice of implementation.
2. One of the shortcomings of computer science has been a resistance to the widespread adoption of formal notation to represent key concepts. For example, in CS 1 it is common to describe the semantics of Pascal statements using natural language. However, natural language can be imprecise and highly ambiguous, and its use can lead to inconsistencies, misinterpretations, and incorrect implementations. Just as the freshman physics major learns to use calculus to express fundamental laws of physics, so must the CS student learn to use precise notation to express the "laws" of computer science. Therefore, to describe the behavior of operations on an abstract data type, we use the method of *axiomatic semantics,* which is a powerful, albeit complex, technique for describing the actions of operations on data objects. While this text contains only an introduction (with additional coverage coming in advanced courses in compilers and formal languages), it should make the student aware of the importance of precise and accurate notation. In addition to axiomatic semantics, we also use Big-O notation for the time and space analysis of algorithms (chapter 6), decision tables and finite state automata for the specification

of software (chapter 13), and a directed graph representation of program design documents (chapter 14).

3. There are many different data structures that can be covered in a CS 2 course, from arrays to AVL trees, from heaps to hash tables. Because of the large number of topics that can be studied, any treatment of this subject should initially provide an overall structure or classification scheme. If this classification is done well, the student will not view the subsequent discussion as a large and apparently unrelated collection of facts but as an organized and related body of knowledge. (The problem is identical to the one faced by a biologist teaching a first course. He or she first presents the well-known taxonomy of living organisms (kingdom, phylum, class, order, . . .) rather than immediately presenting a "Noah's Ark" of animal types. This latter approach would quickly collapse into chaos.) In chapter 3, before beginning our discussion of data structures, we present a taxonomy of data structures based on the following four classes: sets, linear types, hierarchical types, and graphs (Figure 3.1). We then show why all data structures, regardless of what properties they may possess, fall into one of these four classes, thus giving the student a much better idea of how these ideas fit together (the "big picture").

4. Finally, and most controversially, we made the decision not to use a specific high-level language to present our programming examples. Instead, we have chosen to use the pseudocode shown in Figure 2–16. While this pseudocode is closely related to a number of existing languages (especially Modula-2 and Ada), it provides these capabilities in a much more simplified manner. There will no doubt be those who ask why we did not select ———, where the missing word is the name of the individual's favorite language, such as Modula-2, Ada, Turbo Pascal, or C++. To completely and accurately describe the abstract data type capabilities of a "real" language, such as Ada packages, would take many pages of language- and system-specific information. This material would dwell on highly technical syntactic and semantic details such as semicolons, indentation, reserved words, and programming environments. But this approach is exactly opposite from what we stress in this book—that the ideas and concepts we are presenting are language-, system-, and vendor-independent, and that one can study them at a high level of abstraction without worrying about the specific language into which they will be translated (the very definition of abstraction!). This is not a book on advanced concepts in Modula-2 or Ada or ——— but a text on advanced concepts in computer science.

However, computer science is an applied as well as a theoretical discipline, and it is important for a student to implement the abstract ideas presented in a course. It is through this hands-on experience that a student can gain a deeper understanding of these concepts. To support this empirical experience, we have provided a series of language-specific supplements that are coordinated with the material in the text. These supplements describe an actual high-level programming language that can then be used to implement the examples, case studies, and homework assignments. Alternatively, an instructor could choose to provide this language- and system-related information through reference manuals, other texts, or locally prepared handouts. Regardless of the approach, however, it is important to send a message to the student that this is *not* just a language course or a programming course, but a course in computer science. The language that will ultimately be used, so important in CS 1, is more properly viewed in CS 2 as material supplementary to the central discussion.

Part II of the text, chapters 12 to 15, is an overview of the software life cycle. It focuses on those life cycle phases not stressed in the student's previous class work. Four chapters cannot possibly cover software development in great detail, and this text does not purport to replace a full semester course on the subject of software engineering. However, many students graduate in computer science without completing such a course and with little or no introduction to the life cycle of a software project. Part II of the text provides that needed introduction.

In chapter 12 we overview the eight stages in the software life cycle, from the initial rough problem statement to finished documentation and ongoing maintenance. Some of these eight stages may be inappropriate for extended discussion in a CS 2 course, for example, the feasibility study can require a knowledge of economics, accounting, and marketing as well as computer science. Some software development stages have been studied extensively in CS 1, namely coding and debugging. However, the remaining steps in the software life cycle are both highly appropriate to discuss and generally unfamiliar to students in a second course. They are presented in the next three chapters.

Chapter 13 discusses the topics of problem specification and the content of the problem specification document, including functional, error, and performance specifications. As we mentioned earlier, natural language is not the most precise way to express user needs. Therefore, in this chapter we look at two formal alternatives to natural language—decision tables and finite-state automata. These methods can provide the precision needed to develop high-quality specifications.

Chapter 14 treats the program design phase. Here we use the top-down problem-solving strategy called *divide and conquer* to select and specify the program units and data structures needed in our solution. The relationships between these program units will be expressed as a structure chart using a formal notation based on directed graphs. This chapter also describes the information contained in the program design document and presents guidelines for evaluating the quality of a proposed design.

Finally, chapter 15 looks at the two validation and verification (V&V) techniques called *empirical testing* and *formal verification*. We review the strengths and weaknesses of each of these approaches and conclude by describing a third method called *structured walkthroughs*. Regardless of which method is used, the goal of all V&V techniques is the same—to increase the confidence with which we can assert the correctness of each individual program unit as well as the correctness of the software system as a whole.

This is the outline of the text. As the preface clearly shows, it covers a good deal of material and ranges over many subject areas. However, its central themes can be clearly and simply stated: To improve the students' ability to specify, design, and solve real-world problems and to deepen their understanding and appreciation of the discipline of computer science.

Acknowledgments

I would like to thank the people who assisted me during the writing of this text. This certainly includes all the reviewers who read the initial drafts so carefully and made

many helpful suggestions and comments. Their ideas were invaluable to me during the rewriting and revising of the manuscript. Alphabetically, these reviewers were

Douglas Bickerstaff
Eastern Washington University

Thomas E. Byther
University of Maine

Maurice Eggen
Trinity University

Henry Etlinger
Rochester Institute of Technology

Charles E. Frank
Northern Kentucky University

Michael Hennessy
University of Oregon

Lawrence A. Jehn
University of Dayton

Kenneth Modesitt
Western Kentucky University

William R. Nico
California State University at Hayward

Alex T. Prengel, Jr.
Brandeis University

C. Ray Russell
Virginia Commonwealth University

Paul S. Schnare
Eastern Kentucky University

Greg Scragg
State University of New York at Geneseo

Theodore Sjoerdsma
Washington and Lee University

Jeffrey Slomka
Southwest Texas State University

Greg Starling
University of Arkansas, Fayetteville

Stan Thomas
Wake Forest University

Susan R. Wallace
University of North Florida

A special thanks to Mr. David Sielaff who helped prepare and debug all the algorithms and code in the text and I am deeply grateful. Thanks also to Ms. Michelle Villinski who typed the original manuscript and did an outstanding job. Finally, I would like to thank the people at West Publishing who worked with me during the preparation and production of the book. They are professionals in every sense of the word, and they were a pleasure to work with: Mark Jacobsen, Denis Ralling, and, most of all, Jerry Westby, whom I consider to be not only my editor but also a good friend.

G. Michael Schneider
Macalester College

In addition to those named above, I would like to thank Pam Betzel, Jean Bartley, and Wen Zhang.

S.C. Bruell
University of Iowa
August, 1990

CONTENTS

CHAPTER 1

AN INTRODUCTION TO ADVANCED PROGRAMMING CONCEPTS

CHAPTER OUTLINE