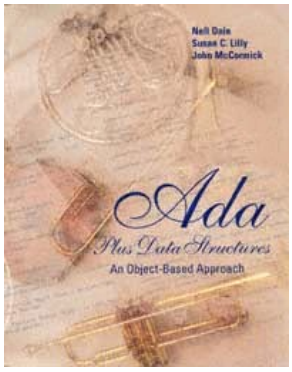
Nell Dale
Susan C. Lilly
John McCormick
Ada
Plus Data Structures
An Object-Based Approach

Ada Plus Data Structures

An Object Based Approach

Nell Dale

The University of Texas at Austin

Susan C. Lilly

IBM

John McCormick

State University of New York at Plattsburgh

D. C. Heath and Company
Lexington, Massachusetts Toronto

Address editorial correspondence to:
D. C. Heath and Company
125 Spring Street
Lexington, MA 02173

Acquisitions: Walter Cunningham

Development: Karen H. Jolie

Editorial Production: Janice Molloy

Cover design: Jan Shapiro

Production Coordination: Charles Dutton

Cover: Lois Ellen Frank/Westlight

Published simultaneously in Canada.

Printed in the United States of America.

International Standard Book Number: 0-669-29265-6

Library of Congress Catalog Card Number: 95-68037

10 9 8 7 6 5 4 3 2 1

This book is dedicated to you,
and to all of our students for whom it was begun and
without whom it could never have been completed.
N. D. S. L. J. M.
To Naomi.
J. M.

Preface

In the past decade computer science has seen software engineering techniques put into practice to solve larger and more complex computing problems than ever before. By merging the study of data structures and algorithms with the related concept of data abstraction, *Pascal Plus Data Structures, Algorithms, and Advanced Programming* was always on the leading edge of computer science education and practice. With *Ada Plus Data Structures: An Object-Based Approach*, we present the next generation of software practices to the first-year student—the object paradigm.

The object paradigm is the latest step in the evolution of software development methodologies. Based on a foundation of abstraction and information hiding, the object paradigm uses a loosely coupled set of objects that interact with each other to model reality more easily. The object class extends the power of the traditional abstract data type to facilitate software design, implementation, maintenance, and reuse.

The object paradigm is still maturing. There is a good deal of debate among software engineers on the best ways to apply object technology and even more debate among educators on how to teach it. We offer an introduction to the object paradigm balanced by a presentation of traditional abstract data types, computer science theory, and concepts derived from software engineering practices. Our approach provides a strong foundation on which students can build in future courses.

An understanding of theoretical concepts helps students put new ideas that they encounter into place, and practical advice allows them to apply what they have learned. To teach these concepts to students who may not have completed college-level mathematics courses, we have used intuitive explanations, even for topics that have a basis in mathematics, such as the analysis of algorithms. In all cases, our highest goal has been to make our explanations as readable and as easily understandable as our programs.

Three Levels of Abstraction

The recurring theme throughout this book is *modeling with levels of abstraction*. Each traditional abstract data type is viewed as an object class from three different perspectives. The first perspective is from the *abstract level*. At this level we deal with the "what" questions. What is the class? What operations can it perform? What exceptions might an operation generate? At the *application level* we explore ways of modeling real-life objects in a specific context. We develop applications using a particular class. Finally, at the *implementation level* we look at specific representations of the class, its accessing operations, and its implementation in Ada. We develop different implementations and explain how to compare them.

We apply the principles of information hiding in our order of presentation. Students concentrate on the *use* of a class in the solution to a problem before implementation of the class is discussed. For example, when developing a post-fix calculator in Chapter 6, students cannot think of the stack as an array or a linked list; they have not yet seen any implementation of the stack class. Their only view of it is at the abstract level. With this order of presentation, students see that the traditional abstract data types are components used in the development of application software rather than as exercises in writing code for their professor. Ada's separation of specification and implementation allows students to code and execute solutions that use an abstract data type without any knowledge of its implementation.

The Ada Language and the Object Paradigm

Standard Pascal offers little support for today's software practices. We have selected Ada as the best language for teaching modern software development methods. Three major strengths of Ada for first-year students are its readability, stability, and reliability. Ada compiler validation ensures that all compilers support exactly the same language—a student can compile any example in this textbook on any platform with any compiler without making modifications to the code. Ada's strong typing and related features ensure that most errors are detected at compile time or through run-time constraints. We believe that the discipline enforced by the Ada compiler is the best tool for preparing students for future courses and industry where less restrictive programming languages are often used.

Ada 95 is an ISO and ANSI standard. It is the first object-oriented language with an international standard. Ada 95 provides support for objects, classes, inheritance, and runtime polymorphism. We have, however, elected to implement our designs in Ada 83. At the time we wrote our code, the Ada 9X compilers, like the language reference manual, were still evolving. Even as we write this preface in the first month of 1995, a complete Ada 95 compiler is not available. Although we refuse to present code to first-year students that might not compile on their systems, we do not ignore Ada 95. The features of Ada 95 that simplify our Ada 83 implementations are discussed in the appropriate places. It is an easy matter for an instructor to substitute Ada 95 constructs where appropriate.

We take an object-based approach (object-oriented without inheritance and run-time polymorphism) rather than an object-oriented approach for both pragmatic and pedagogical reasons. As mentioned above, at the time of this writing a complete Ada 95 compiler was not available. More important, we believe that the complexity that inheritance adds to a first-year student's learning process exceeds the reduction in design complexity made possible by it. Gaining a thorough understanding of objects, classes, and static polymorphism through an object-based approach provides a solid base for the development of larger projects where the benefits of inheritance and run-time polymorphism provided by an object-oriented approach are best appreciated.

Prerequisite Knowledge

We assume that the student reader has an elementary knowledge of the Ada programming language that includes the following topics:

1. Basic declarations (including predefined and programmer-defined scalar types, record types, and array types)

2. Input and output using package `Text_IO` and its generic packages (`Float_IO, Integer_IO`, and `Enumeration_IO`)

3. Basic control structures

4. Subprograms (functions and procedures)

5. How to use resources available in predefined and instructor-defined packages

The material in Appendix L is designed to allow students with previous Pascal experience to quickly master the Ada basics needed to use this book effectively.

Content and Organization

Chapters 1 through 3 present a breadth-first approach to software engineering. In Chapter 1 we discuss the basic goals of high-quality software and the basic principles of software engineering for designing and implementing programs to meet these goals. We review functional design techniques and introduce object-oriented design. Modularization, good programming style, documentation, and the separation of the design of a problem solution from its implementation are stressed throughout this chapter. We introduce packages as a method for implementing an object-oriented design. Finally, because there is more than one way to solve a problem, we discuss how competing solutions can be compared through the analysis of algorithms, using Big-O notation.

Chapter 2 addresses what we see as a critical need in software education—the ability to design and implement correct programs and to verify that they are actually correct. Topics covered in this chapter include the concept of "life-cycle" verification; designing for correctness using preconditions, postconditions, and loop invariants; the use of deskchecking and design/code walkthroughs and inspections to identify errors before testing; debugging techniques, data coverage (black box), and code coverage (clear or white box) approaches; unit testing, test plans, and structured integration testing using stubs and drivers. A case study shows how all these concepts can be applied to the development of a binary search procedure.

Chapter 3 begins our study of class design and implementation. We introduce the three perspectives of object classes: abstraction, application, and implementation. Then we apply these perspectives to some of Ada's predefined classes that students learned in their introductory programming course: scalar types, records, and arrays. We introduce data structures (collection classes) with a real-world example (a library). We illustrate how Ada packages are used for information hiding, introduce private types as a means of encapsulation, and explain why information hiding and encapsulation are important. We conclude the chapter with a case study that illustrates how the three levels of abstraction are used in the design and implementation of a Bingo game simulation.

In Chapter 4 we introduce the discrete set class. The set is first considered from its abstract perspective that we describe formally through a package specification. Then, before discussing any implementation details, we use the set specification to implement the Bingo Basket object used in the previous chapter's case study. At this point, students can design and implement software that *uses* the set class. Only then do we discuss the third perspective—the implementation of the discrete set class. Chapter 4 concludes with a case study that uses the set class in a solution to a realistic problem. We also use this case study to introduce the nature of a greedy solution.

Chapter 5 discusses the string class. We begin with the terminology and classification of strings: fixed-length, varying-length, bounded-length, and unbounded-length. We present a string class specification, implement a simple application that uses it, and develop a bounded-length implementation. We use the unbounded-length string class to motivate the need for dynamic allocation. We introduce the concept of dynamic allocation along with

the syntax for using Ada's access types. Our implementation of the unbounded-length string class demonstrates the value of pointers without having to introduce the additional complexities of linked structures.

Chapter 6 introduces the stack class. Again our order of presentation is abstract level (package specification), application level (a program to evaluate postfix expressions), and implementation level (package body). We present two implementations of the stack class: array-based and linked. The technique used to link nodes in dynamically allocated storage is described in detail and illustrated with figures. We analyze the two stack representations in terms of their number of source lines, their use of storage space, the efficiency of their operations using Big-O notation, and timed test runs. Finally, we revisit Ada's encapsulation mechanisms (introduced in Chapter 3) in the light of our stack class.

In Chapter 7 we introduce the FIFO queue. After specifying the FIFO queue class we develop an application to prepare freight train manifests. We use this application to review our object-oriented approach to software development. In this chapter we give a detailed look at the design considerations of selecting among multiple implementation choices. Two array-based implementations are discussed, as well as a linked queue representation. We analyze our FIFO queue implementations in terms of their number of source lines, their use of storage space, the efficiency of their operations using Big-O notation, and timed test runs. Finally, we examine techniques for testing the queue operations using a test driver.

Chapter 8 introduces linear lists that are ordered according to key value. In developing the abstract view of ordered lists we introduce the three operator classes: constructors, observers, and iterators. We use our list class specification in the design and implementation of an electronic address book. Our program uses retained data from its previous execution, which is stored in a binary file. We explain the use of sequential binary files for students who have never used them. The list class is implemented using both sequential (arraybased) and linked (dynamically allocated) representations. These two implementations are compared in detail, in terms of the size of their source code, their storage requirements, Big-O analysis of their operations, and timed test runs.

Chapter 9 continues the discussion of linked lists with a number of implementation variations: linked lists with dummy nodes (headers and trailers), circular linked lists, and doubly linked lists. The insertion, deletion, and list traversal algorithms are developed and implemented for each variation. We compare the different implementations in terms of the size of their source code, their storage requirements, Big-O analysis of their operations, and timed test runs. We conclude this chapter with a case study—the design and implementation of an unbounded natural number class.

In Chapter 10 we present alternative approaches for the storage of linked structures. First we discuss why we might want to store linked structures in an array rather than in dynamic memory. Then we modify the linked implementation of the ordered list developed in Chapter 8 to store the linked lists in an array rather than in dynamic memory. We compare the array-based and heap-based linked list implementations in terms of Big-O and timed test runs. Finally we show how direct files can be used to store linked lists that are retained between program runs. We explain direct files for students who have never used them and present a complete direct file-based linked list implementation.

In Chapter 11, our discussion of recursion gives students an intuitive understanding of the concept and then shows them how they can use recursion to solve programming problems. This chapter is suitable for introducing *or* reviewing recursion. Guidelines for writing recursive procedures and functions are illustrated with many examples. After demonstrating that by-hand simulation of a recursive routine can be very tedious, we introduce a

simple three-question technique for verifying the correctness of recursive procedures and functions. Because many students are wary of recursion, the introduction to this material is deliberately intuitive and nonmathematical. A more detailed discussion of how recursion works leads to an understanding of how recursion can be replaced with iteration and stacks. The case study at the end of this chapter is a recursive solution of a maze problem. We compare this implementation to a nonrecursive (stack-based) approach to demonstrate how recursion can simplify the solution to some kinds of problems.

Chapter 12 introduces binary search trees as a way to arrange data, giving the flexibility of a linked structure without the liability of slow, sequential access to its elements. We develop the tree operations in detail, then implement them using dynamic allocation with pointer (access type) variables. We present both recursive and nonrecursive versions of the insertion, deletion, retrieve, and modify operations, reinforcing the use of recursion for simplifying programming problems.

Chapter 13 presents a collection of other branching structures heaps, priority queues (implemented with heaps), and graphs. The coverage of graphs includes the specification of a Graph class, its use in an airline application (connections between cities), and the implementation of basic graph operations with an adjacency matrix. We create both depth first and breadth-first search procedures using the Graph, Stack, and FIFO Queue classes. A procedure is developed that displays the shortest paths to all nodes from a single source using the Graph and Priority Queue classes. The chapter also describes and illustrates the use of adjacency list graph representations.

Chapter 14 presents a number of sorting algorithms and asks the question. Which is best? The sorting algorithms that are illustrated, implemented (as generic procedures) and compared include the straight selection sort, two versions of the bubble sort, insertion sort. merge sort, quick sort, and heap sort. We analyze the storage requirements of each of the sorting algorithms and efficiency in terms of both Big-O and timed test runs. Finally, to illustrate a completely different kind of sorting algorithm (one that does not compare keys), we design and implement a generic radix sort procedure.

Chapter 15 continues the discussion of algorithm analysis in the context of searching. Various searching algorithms are explained, implemented, and compared, including the sequential and binary searches developed and used earlier in the book. We introduce hashing and discuss the most common hash functions and collision resolution techniques.

Additional Features

Chapter Goals

A set of goals is presented at the beginning of each chapter to help students assess what they have learned. These goals are tested in the exercises at the end of each chapter.

Case Studies

In most chapters we develop an application program to illustrate the use of a particular class. These applications are short enough that students can follow them without being overwhelmed by a multitude of details. In addition to these application programs, we present six larger-scale case studies. Program reading is an essential skill for software professionals, but few books include programs of sufficient length for students to get this experience. The case studies provide an opportunity to follow the specification, design, and implementation of a solution to a nontrivial problem. They also provide a base for class discussion on design issues and programming assignments.

Complete Code

We include all of the Ada code for each class discussed in the text—both in print and on the program disk provided with the text. Having the source code for the abstract data types on disk encourages students to think in terms of reusable code. The source code for the application programs and case studies is provided to give students practice in modifying programs without having to spend time rekeying the original program.

Chapter Exercises

Most chapters have more than 30 exercises. The exercises have varying levels of difficulty, including short coding problems, the analysis of algorithms, and problems that test students' understanding of concepts. For chapters that contain case studies, there are exercises that specifically pertain to the material in the case study. These exercises are designed to motivate students to read the case studies carefully. Approximately one-third of the exercises are answered in the back of the text; the answer key for the remaining exercises is in the *Instructor's Guide*.

Programming Assignments

A set of programming assignments for each chapter is included at the end of each chapter. The assignments represent a range of difficulty levels and were carefully chosen to illustrate the techniques described in the text. These assignments, which include modifications and enhancements to the application programs and case studies, give students experience in program modification and program "maintenance." A further selection of programming assignments is available in the *Instructor's Guide*.

Instructor's Guide

An *Instructor's Guide* is available with suggestions for how to teach the material covered in each chapter, a key to answers to the chapter exercises not in the book, a set of questions for creating tests, and transparency masters for classroom teaching.

Acknowledgments

It is a pleasure to be able to thank the many people who helped us turn a list of good ideas into a real book. We have been extremely lucky to have many technical reviewers whose comments, corrections, and suggestions have enormously enriched this book. The successful completion of this text is largely due to the scrupulous attention of our eagle-eyed reviewers: John C. Arch, Shippensburg University; Jack Beidler, University of Scranton; Ronald L. Carlisle, Oglethorpe University; Jose Cordova, Southern Arkansas University; John Crenshaw, Western Kentucky University; Ernest Ferguson, Southwest Baptist University; Charles E. Frank, Northern Kentucky University; George C. Harrison, Norfolk State University; Charles C. Kirkpatrick, Parks College of Saint Louis University; Artur J. Kowalski, New Jersey Institute of Technology; Bill Kraynek, Florida International University; Herbert Mapes, Gallaudet University; James McCaffrey, Hawaii Pacific University; Melvin Neville, Northern Arizona University; Thomas M. Phillips, Auburn University; Paul D. Phillips, Mt. Mercy College; David A. Retterer, Ohio Northern University; Scott Sigman, Southwest Baptist University; Robert Steigerwald, US Air Force Academy; Ricky E. Sward, US Air Force Academy; Curt M. White, Indiana University-Purdue University, Fort Wayne; and Phyllis Williams, East Stroudsburg University.

We are indebted to the students enrolled in CSC 223 at the State University of New York at Plattsburgh for enduring nearly two years of working with an evolving manuscript. Their comments were invaluable in making this book accessibleto future students.

The staff at D. C. Heath has been extraordinarily helpful. We particularly thank our developmental editor, Karen Jolie, for her dedication to this project and the encouragement

she provided us while doing the job of three people. We also thank editorial assistant Heather Monahan and acquisitions editor Randall Adams. Finally, we thank our production editor, Janice Molloy, who was responsible for turning a very large stack of manuscript pages into a beautiful book.

Anyone who has ever written a textbook knows the amount of time and effort that goes into such a project. Anyone who is related to a textbook author can tell you at whose expense that time is spent. Many thanks to Naomi who, while writing her own book, cheerfully shared the computing facilities in the McCormick house.

N. D.
S. L.
J. M.

Contents