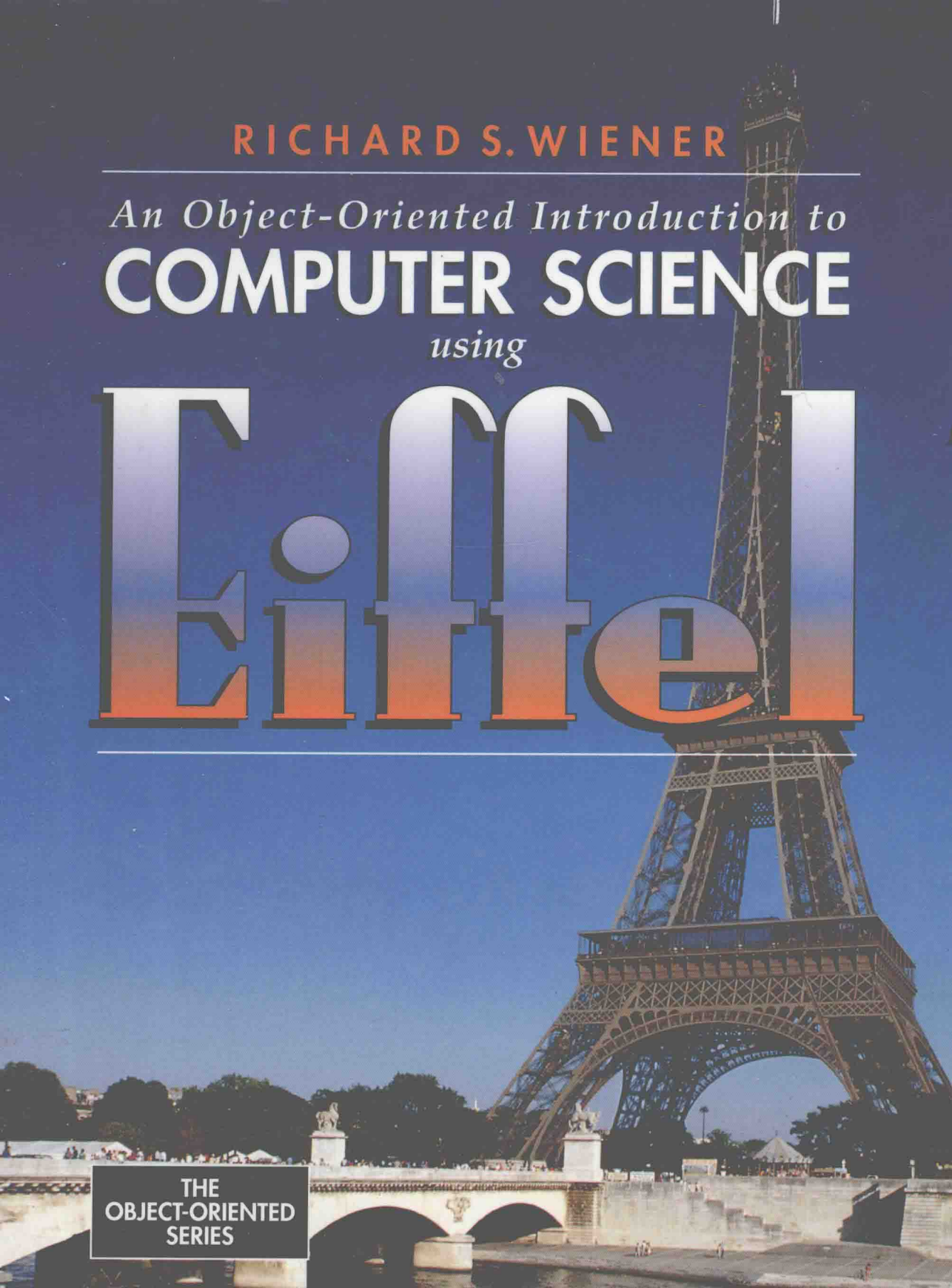

RICHARD S. WIENER
An Object-Oriented Introduction to
COMPUTER SCIENCE
using
Eiffel
THE OBJECT-ORIENTED SERIES

An Object-Oriented Introduction to Computer Science Using Eiffel

Richard S. Wiener

Prentice Hall PTR
Upper Saddle River, New Jersey 07458

Library of Congress Cataloging-in-Publication Data
Wiener, Richard, 1941–
An object-oriented introduction to computer science using Eiffel / by Richard S. Wiener
p. cm. -- (Prentice Hall object-oriented series)
Includes index.
ISBN 0-13-183872-5
1. Object-oriented programming (Computer science) 2. Eiffel (Computer program language 3. Computer science.. I. Title. II. Series.
QA76.64.W44 1996
005.13'3--dc20 96-2186
CIP

Editorial/production supervision and Interior Design: *Joanne Anzalone*
Manufacturing manager: *Alexis R. Heydt*
Acquisitions editor: *Paul Becker*
Editorial assistant: *Maureen Diana*
Cover design: *Design Source*
Cover design director: *Jerry Votta*

Prentice-Hall, Inc.
A Simon & Schuster Company
Upper Saddle River, New Jersey 07458

The publisher offers discounts on this book when ordered in bulk quantities.
For more information, contact:
Corporate Sales Department
Prentice Hall PTR
1 Lake Street
Upper Saddle River, NJ 07458

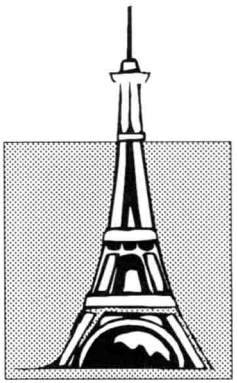
Phone: 800-382-3419, Fax: 201-236-7141
E-mail: corpsales@prenhall.com

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 0-13-183872-5

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*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

This book is dedicated with all my love to my sons
Henrik, Marc, and Erik.



Preface

There is a strong need for a CS 1 book that from the very beginning presents the basic principles of computer science from an object-oriented perspective and is supported by a friendly, consistent, and relatively easy to learn object-oriented programming language. An object-oriented perspective represents a further evolution in the trend to emphasize abstractions in computer problem solving and the use of abstract data types in particular in early computer science courses.

This book is aimed at the beginning computer science student enrolled in a rigorous computer science curriculum. It is also aimed at practicing software development professionals new to the object paradigm who wish a gentle introduction to many features of the Eiffel language and the object paradigm.

This book presents the basic ideas of object modeling from the very beginning. Before a student learns to "program," he or she should be introduced to modeling. It is important that the beginning student as well as practicing software development professionals view programming as only part of the intellectual process associated with software development and computer science. Booch class and object scenario diagrams are introduced early as a means of providing notational support and more importantly support for the notion of object modeling.

The object-oriented perspective is quite distinct from the older traditional approach of having students learn the rudiments of programming from the bottom up. That is, first learn about scalar types, variables,

assignment operations, branch and loop program control structures, and much later the concept of functional abstraction. Although in recent years functions have been introduced earlier in some CS 1 books, it is often the case that they are first introduced in the middle of the book.

Using an object-oriented perspective, functions and the underlying data model that they are manipulating are introduced from the very beginning. The class is introduced early as a frame from which to introduce and implement simple algorithms and provide a model for objects.

Some computer science departments have been moving towards C or C++ to support CS 1. This author believes that this is a grave mistake. Although both of these languages are commercially important and widely used outside of the university, which probably accounts for their adoption as a CS 1 language, they are poor candidates to support CS 1. Both languages are complex, are relatively hard to read, provide relatively little safety to the beginning programmer, and are relatively inconsistent (particularly C++). They both require the student to take a fairly low-level systems view quite early. It therefore becomes quite challenging for the beginning student to master low-level details and at the same time develop a high-level vision and sensitivity concerning the safe construction of software systems. The Eiffel language is much better suited for this task.

Eiffel is quite readable, friendly, and consistent. The dangerous artifact of pointers is totally missing. Memory management is handled automatically. Eiffel's assertion handling mechanism provides an opportunity to emphasize safe and defensive programming. Its clean and simple syntax and semantics for handling generic components, late-binding, and inheritance allow a student to focus on the fundamental concepts of software construction and algorithm design without having to become distracted with the myriad of complex language details required, for example, if one uses C++.

Chapter 1 provides a short historical perspective related to computation and computers.

Chapter 2 introduces the concept of objects and object modeling. Objects as abstractions of reality are presented. The noun-verb metaphor, the notion of state, object scenarios and messages, classification, inheritance, aggregation and the uses relationship are introduced. An introduction to object-oriented programming is provided through a simple example. Some of the Booch analysis and design notation and the concepts behind the notation are introduced.

Chapter 3 introduces the reader to the world of programming using Eiffel. The basic elements of an Eiffel software system are presented. These

include creating and destroying objects, basic types, reference versus value semantics, object assignment, object copying, object cloning, branching, iteration, and the construction of routines. In addition the use of basic Eiffel libraries is introduced.

Chapter 4 focuses on the design of algorithms. A graduated set of problems of increasing complexity are used to illustrate the rudiments of algorithm design and develop sensitivity to algorithm complexity.

Chapter 5 presents the reader with some first examples of complete Eiffel software systems. A preview is provided concerning the use of inheritance, late-binding, and assertions. A pair of ordinary dice are simulated. Then a pair of unusual non-standard dice are constructed using inheritance. A race horse game to be played by a person against the computer is built that uses the non-standard dice. Finally, a counterfeit coin weighing game is created that allows a person to play with the assistance of the computer.

Chapter 6, "The Construction of Eiffel Classes," presents more detail related to the various sections of an Eiffel class and their use. Object creation, routine redefinition and renaming, and export scope are among the topics covered. The important facility of assertion handling is presented in this chapter.

Chapter 7 discusses the issue of building reusable container classes. Several classic container classes are presented including STACK, QUEUE, UNORDERED_LIST, ORDERED_LIST, DEQUE, and SET. The BIT data type is introduced and used as part of the implementation of SET.

Chapter 8 introduces recursion as a design technique. First the mechanics of recursion are presented. The relationship between recursion and iteration is discussed and illustrated. Several smaller examples that illustrate recursive designs are presented including binary search of an array and quicksort. The chapter ends with an intermediate sized example involving a depth-first search of a graph. The reader is introduced to the flavor of more advanced algorithm design, an important foundation subject in computer science.

Chapter 9 presents polymorphism and late-binding as a design principle. After illustrating the principle with a simple and somewhat sterile example, an initial and improved version involving the analysis, design, and implementation of a complete software system are presented. Booch class and object scenario diagrams are used to support the analysis and design.

Acknowledgments

I would first like to thank Paul Becker, publisher at Prentice Hall, for his support and encouragement from this project's inception to its completion.

I am in debt to several outstanding reviewers who have provided extremely useful and constructive criticism of the first-draft manuscript.

Jim McKim of the Hartford Graduate Center, friend, Eiffel mentor, and outstanding critic, has examined every line of code in this manuscript and has made many useful suggestions. As before, Jim, my simple words of thanks are really not enough to thank you for your efforts way above and beyond the call of duty. The entire Eiffel community owes you many thanks for the continuing contributions that you are making.

Brian Henderson Seller, from the University of Technology in Sydney, has provided many helpful comments, particularly regarding the sections of the book dealing with object modeling.

Meilir Page Jones, President of Wayland Systems, has provided tremendous help in his critical but extremely constructive review of the manuscript. His many annotations in the first-draft manuscript have provided significant help in improving the book.

I am particularly appreciative of the timely help provided by Jim, Brian, and Meilir because I know how busy they are. Thank you all for finding the time to fit this manuscript review into your busy schedules.

I thank Margaret Reek for looking at a near final version of the manuscript and providing useful and constructive comments.

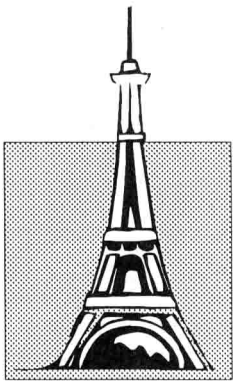
I wish to thank Interactive Software Engineering in Santa Barbara for continuing to provide me with their latest Eiffel software. It is my hope that the Professional Version of Eiffel for MSDOS/Windows will make this elegant language much more accessible to students and professionals alike.

I wish to thank Bertrand Meyer, the original designer and implementor of Eiffel, for his encouragement and support.

I also wish to thank Rock Howard and Madison Cloutier and everyone at Tower Technology for their technical support, tremendous encouragement and latest Eiffel products. Their outstanding contributions to the Eiffel community are noteworthy.

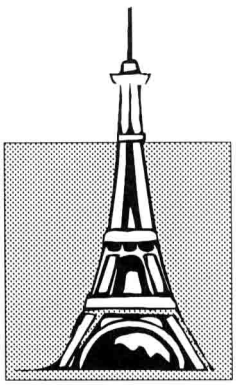
With great love and appreciation, I thank my wife Hanne for her help, constructive criticism, and continual encouragement.

Richard Wiener



Contents

Chapter 1



Programming and Software

1.1 Computer science

Many readers of this book may be enrolled in their first computer science course. Welcome to computer science! Other readers may be wishing to learn more about object-oriented software development. Welcome to this exciting paradigm! (The word paradigm means "a set of forms all of which contain a particular element" —*Random House Dictionary.*)

Typically a first course in computer science introduces a programming language and focuses on programming. Some students may leave a CS 1 course with the impression that computer science is the study of programming. This is not true.

Software is the end product of an engineering process that involves requirements, specifications, analysis, and design. Software is a tangible and visible entity. It is the instructions that permit a digital computer to perform a variety of tasks. Software is a product often shrink-wrapped with a fancy cover. Software is a multibillion dollar business.

A programming language provides a notation in which to express algorithms and information structures. Reasoning can be done with this notation. But to many computer scientists, programs represent the least creative, most routine, and perhaps most tedious part of the software development process. In fact, some computer scientists do not even program.

To other computer scientists, the creation of programs and software systems is what computer science is all about. The theory of programming languages underscores the importance of programming. But computer science is much more than programming.

Computer science deals with the art, craft, and science of computation using a digital computer. Computer science is a theoretical as well as practical discipline; a theoretical as well as applied science. The theory of automata, artificial and natural languages, learning and cognition, information, data structures, complexity, and algorithms play a central role and serve as a theoretical underpinning for all of computer science. The major application areas of computer science include operating systems, compiler design, data structures and algorithms, graphics, numerical analysis, databases, programming languages, artificial intelligence, machine learning, and software engineering. As a computer science student, you will be required to take courses in many or all of these areas.

Most applied sciences require their practitioners to express their ideas in one or more technical languages. Chemists learn the language of chemical symbols and the operators and connectors that allow chemical equations to be written. Physicists use the language of calculus, differential equations, and other advanced mathematics to express their models and their ideas. Electrical engineers learn the language of circuit diagrams. Computer scientists also use a variety of notations and languages to express their concepts and produce their results.

A physics student must first learn some basic mathematics in order to have a notation that can be used for discussion and reasoning about physics. A computer science student needs to learn a high-level programming language and problem solving techniques in order to be able to reason about computation. Programming no more defines what a computer scientist does than calculus defines what a physicist does.

Computer scientists, like their natural science and engineering colleagues, are concerned with model building, abstractions, analysis, design, and implementation. A program or software system often represents the final step in a reasoning and problem solving process.

This book will introduce techniques for reasoning and problem solving using objects. The fundamental principles of object-oriented programming will be explored and introduced. Through this exploration, many important principles of computation will be revealed.

1.2 Computer programs

A program consists of a sequence of instructions written in a precisely defined language called a programming language. These instructions are translated by a compiler into a low-level language, machine language, that the computer can respond to.

Software applications are generally divided into two broad categories: systems programs and applications programs. System programs are aimed at controlling a computer component such as a storage device, output device, or the computer itself (e.g., operating system). Application programs solve a specific problem external to the computer such as a banking application, air-traffic control system, word processing system, spreadsheet, or some other application area.

Computer programs represent the end product of the software development process. They are tangible entities that can be delivered to a customer, billed for, and shrink-wrapped. Commercial programs usually come packaged with a User's Guide and other supporting written documentation.

1.3 Programming languages

Three broad categories of programming languages have been developed: machine languages, assembly languages, and high-level languages. The earliest computers could be programmed using only a machine language. Such a language uses a sequence of 0's and 1's (bits) that represent precise instructions for computation and data access.

Assembly languages use alphabetic characters (letters) to represent the bit configurations in machine language. The letters usually describe the operations to be performed. Assembly languages represent a higher level of abstraction than a machine language. Some modern assembly languages support control structures that were previously found only in high-level languages.

High-level languages resemble natural languages. Data and operations are represented by descriptive statements.

As an example, suppose we wish to add two numbers and deposit the sum in a third number. In many high-level languages this operation would be symbolized:

```
c := a + b
```