Bertrand Meyer

Object-oriented Software Construction

PRENTICE HALL INTERNATIONAL SERIES IN COMPUTER SCIENCE

C.A.R. HOARE SERIES EDITOR

OBJECT-ORIENTED SOFTWARE CONSTRUCTION

Bertrand Meyer

Interactive Software Engineering,
Santa Barbara, California
and
Société des Outils du Logiciel,
Paris

PRENTICE HALL

NEW YORK LONDON TORONTO SYDNEY TOKYO

Pour Annie, Caroline, Isabelle-Muriel,
Laurent, Raphaël et Sarah

First published 1988 by
Prentice Hall International (UK) Ltd,
66 Wood Lane End, Hemel Hempstead,
Hertfordshire, HP2 4RG
A division of
Simon & Schuster International Group

Printed and bound in Great Britain at
the University Press, Cambridge.

Library of Congress and British Library Cataloging-in-Publication Data are available upon application to the publisher.

3 4 5 92 91 90 89 88

ISBN 0-13-629049-3

ISBN 0-13-629031-0 PBK

OBJECT-ORIENTED SOFTWARE CONSTRUCTION

Prentice Hall International
Series in Computer Science

C. A. R. Hoare, Series Editor

BACKHOUSE, R. C., *Program Construction and Verification*
BACKHOUSE, R. C., *Syntax of Programming Languages: Theory and practice*
DE BAKKER, J. W., *Mathematical Theory of Program Correctness*
BIRD, R., AND WADLER, P., *Introduction to Functional Programming*
BJÖRNER, D., AND JONES, C. B., *Formal Specification and Software Development*
BORNAT, R., *Programming from First Principles*
BUSTARD, D., ELDER, J., AND WELSH, J., *Concurrent Program Structures*
CLARK, K. L., AND MCCABE, F. G., *micro-Prolog: Programming in logic*
DROMEY, R. G., *How to Solve it by Computer*
DUNCAN, F., *Microprocessor Programming and Software Development*
ELDER, J., *Construction of Data Processing Software*
GOLDSCHLAGER, L., AND LISTER, A., *Computer Science: A modern introduction (2nd edn)*
HAYES, I. (ED.), *Specification Case Studies*
HEHNER, E. C. R., *The Logic of Programming*
HENDERSON, P., *Functional Programming: Application and implementation*
HOARE, C. A. R., *Communicating Sequential Processes*
HOARE, C.A.R., AND SHEPHERDSON, J. C. (EDS), *Mathematical Logic and Programming Languages*
INMOS LTD, *occam Programming Manual*
INMOS LTD, *occam 2 Reference Manual*
JACKSON, M. A., *System Development*
JOHNSTON, H., *Learning to Program*
JONES, C. B., *Systematic Software Development using VDM*
JONES, G., *Programming in occam*
JONES, G., *Programming in occam 2*
JOSEPH, M., PRASAD, V. R., AND NATARAJAN, N., *A Multiprocessor Operating System*
LEW, A., *Computer Science: A mathematical introduction*
MACCALLUM, I., *Pascal for the Apple*
MACCALLUM, I., *UCSD Pascal for the IBM PC*
MEYER, B., *Object-oriented Software Construction*
PEYTON JONES, S. L., *The Implementation of Functional Programming Languages*
POMBERGER, G., *Software Engineering and Modula-2*
REYNOLDS, J. C., *The Craft of Programming*
SLOMAN, M., AND KRAMER, J., *Distributed Systems and Computer Networks*
TENNENT, R. D., *Principles of Programming Languages*
WATT, D. A., WICHMANN, B. A., AND FINDLAY, W., *ADA: Language and methodology*
WELSH, J., AND ELDER, J., *Introduction to Modula-2*
WELSH, J., AND ELDER, J., *Introduction to Pascal (2nd edn)*
WELSH, J., ELDER, J., AND BUSTARD, D., *Sequential Program Structures*
WELSH, J., AND HAY, A., *A Model Implementation of Standard Pascal*
WELSH, J., AND MCKEAG, M., *Structured System Programming*
WIKSTRÖM, Å., *Functional Programming using Standard ML*

Preface

Born in the ice-blue waters of the festooned Norwegian coast; amplified (by an aberration of world currents, for which marine geographers have yet to find a suitable explanation) along the much grayer range of the Californian Pacific; viewed by some as a typhoon, by some as a tsunami, and by some as a storm in a teacup – a tidal wave is reaching the shores of the computing world.

"Object-oriented" is the latest *in* term, complementing or perhaps even replacing "structured" as the high-tech version of "good". As is inevitable in such a case, the term is used by different people with different meanings; just as inevitable is the well-known three-step sequence of reactions that meets the introduction of a new methodological principle: (1) "it's trivial"; (2) "besides, it won't work"; (3) "anyway, that's how I did it all along". (The order may vary.)

Let's make it clear right away, lest the reader think the author takes a half-hearted approach to his topic: I do not think object-oriented design is a mere fad; I think it is not trivial (although I shall strive to make it as limpid as I can); I know it works; and I believe it is not only different from but even, to a certain extent, incompatible with the software design methods that most people use today – including some of the principles taught in most programming textbooks. I further believe that object-oriented design has the potential for significantly improving the quality of software, and that it is here to stay. Finally, I hope that as the reader progresses through these pages, he will share some of my excitement about this promising avenue to software design and implementation.

"Avenue to software design and implementation". The view of object-oriented design taken by this book is definitely that of software engineering. Other perspectives are possible: there has been much interest in applying object-oriented methods to Artificial Intelligence, graphics or exploratory programming. Although the presentation

does not exclude these applications, they are not its main emphasis. We study the object-oriented approach as a set of principles, methods and tools which can be instrumental in building "production" software of higher quality than is the norm today.

Object-oriented design is, in its simplest form, based on a seemingly elementary idea. Computing systems perform certain actions on certain objects; to obtain flexible and reusable systems, it is better to base the structure of software on the objects than on the actions.

Once you have said this, you have not really provided a definition, but rather posed a set of problems: What precisely is an object? How do you find and describe the objects? How should programs manipulate objects? What are the possible relations between objects? How does one explore the commonalities that may exist between various kinds of objects? How do these ideas relate to classical software engineering concerns such as correctness, ease of use, efficiency?

Answers to these issues rely on an impressive array of techniques for efficiently producing reusable, extendible and reliable software: inheritance, both in its linear (single) and multiple forms; dynamic binding and polymorphism; a new view of types and type checking; genericity; information hiding; use of assertions; programming by contract; safe exception handling. Efficient implementation techniques have been developed to allow practical application of these ideas.

In the pages that follow, we shall review the methods and techniques of object-oriented software construction. Part 1 (chapters 1 to 4) describes the software engineering issues leading to the object-oriented approach, and the basic concepts of object-oriented design. Part 2 (chapters 5 to 16) reviews object-oriented techniques in detail; this part of the book relies on the object-oriented language **Eiffel**. Part 3 (chapters 17 to 20) looks at the implementation of object-oriented concepts in other environments: classical, non-object-oriented languages such as Fortran, Pascal and C; modular but not really object-oriented languages such as Ada and Modula-2; object-oriented languages other than Eiffel, such as Simula 67 and Smalltalk. Part 3 concludes with a brief review of current issues such as concurrency and persistency. Part 4 contains a number of appendices, particularly on details of Eiffel.

Eiffel plays an important part in this book and its use deserves a few comments. Attempts to discuss issues of software design independently of any notation may seem commendable, but are in fact naive, and bound to yield superficial results. Conversely, many discussions of what appear to be language problems are in fact discussions of serious software engineering problems. Object-oriented design is no exception; to describe it thoroughly, one needs a good notation. For me, Eiffel is that notation, which I designed because no existing language was up to my expectations. In other words, Eiffel is used in this book to support the concepts rather than the other way around. My estimate is that 90% of the material will be useful to readers interested in object-oriented design, even if they never approach the Eiffel programming environment. The remaining 10% is mainly concentrated in the appendices and syntactic notes at the end of each chapter. Part 3 explains how the concepts may be transposed to other languages.

Some of the chapters of part 2 include a "discussion" section explaining the design issues encountered during the design of Eiffel, and how they were resolved. Being the language designer, I felt this was some of the most useful information I

could try to convey. I hope the reader will see in these discussions not attempts at self-justification, but candid insights into the process of language design, which holds much in common with the process of software design. I often wished, when reading descriptions of well-known programming languages, that the designers had told me not only what solutions they chose, but why they chose them.

The use of a programming notation should not lead the reader to believe that object-oriented techniques only cover the implementation phase. Quite to the contrary, much of this book is about **design**. Software design is sometimes mistakenly viewed as an activity totally secluded from actual implementation. A tendency has even arisen recently to present simple graphical notations, perhaps adequate for *expressing* designs, as "design methods" (or better yet, "methodologies"). In reality, design involves the same intellectual mechanisms and the same intellectual challenges as programming, only at a higher level of abstraction. Much is to be gained from an approach that integrates both activities within the same conceptual framework. Eiffel was conceived with this goal in mind; such language features as deferred classes, information hiding and assertions address it directly. Several chapters (especially 3, 4, 7, 9, 12 and 14) specifically discuss issues of high-level design.

Although I take full responsibility for any flaws in this book and the design of Eiffel, I acknowledge with great pleasure the help received from many people. The foremost influence has been that of Simula, which introduced most of the concepts twenty years ago, and had most of them right; Tony Hoare's remark about Algol 60 – that it was such an improvement over most of its successors – applies to Simula as well. The staff of Interactive Software Engineering helped tremendously. Jean-Marc Nerson contributed numerous insights and implemented some of the tools of the Eiffel environment; his constant support has been decisive. He and Reynald Bouy, as the first Eiffel programmers, provided feedback and suggestions at a crucial time. The first implementation of Eiffel was started by Deniz Yuksel and brought to completion by Olivier Mallet, Frédéric Lalanne and Hervé Templereau; in this process they came up with many brilliant insights, regarding not only implementation techniques but the language itself. Key contributions were also made by Pascal Boosz. The help of Ruth Freestone and Helen Martin from Prentice-Hall International in bringing the manuscript to production was much appreciated. I am also indebted to Peter Lohr, W. Rohdewald and especially David Yost for pointing out a number of errors in the first printing. Finally, I have given short courses and lectures on the topics of this book on three continents, and the participants' questions and comments have considerably enriched my understanding of the field, as have the many suggestions contributed by the commercial and academic users of Eiffel.

Santa Barbara
July 1988

B.M.

Acknowledgments

Some of the material in appendix B appeared in part in "Eiffel: Programming for reusability and extendibility", *SIGPLAN Notices*, 22, 2, February 1987, pp. 85-94.

Some of the material in chapters 3 and 12 appeared in part in "Reusability: The Case for Object-Oriented Design", *IEEE Software*, 4, 2, March 1987, pp. 50-64

Some of the material in chapter 19 appeared in part in "Genericity vs. Inheritance", Proc. OOPSLA Conference, ACM, October 1986; revised version to appear in *Journal of Pascal, Ada and Modula-2*.

Some of the material in chapter 4 and appendix A appears in part in "Eiffel: A Language and Environment for Software Engineering", *The Journal of Systems and Software*, 1988.

Trademarks used in this book: Ada (US Department of Defense); Eiffel (Interactive Software Engineering, Inc.); Objective-C (Productivity Products International); Simula 67 (Simula AS); Smalltalk (Xerox); Unix (AT&T Bell Laboratories).

Author's addresses

Interactive Software Engineering, Inc.
270 Storke Road, Suite 7 Goleta, CA 93117 - USA

Société des Outils du Logiciel
Centre d'Affaires 3 MPP, 4 rue René Barthélémy 92120 Montrouge - France

Syntax notation

The following notation, a simple variant of BNF (Backus-Naur Form), is used in the syntactical descriptions found at the end of each chapter on Eiffel and in Appendix C.

Language structures are defined as "constructs", whose names start with a capital letter and are written in normal (roman) font, as Class, Instruction etc. The syntactical form of the instances of a construct is given by a production of the form:

Construct = Right_hand_side

Every syntactical construct appears on the left-hand side of exactly one production, except for the lexical constructs (Identifier, etc.) which are defined separately.

The right-hand side of a production is a sequence of constructs and/or terminals, where a terminal represents an actual language element (keyword such as **class**, operator such as +, etc.). Terminals are written as follows:

- Keywords appear in boldface and stand for themselves, for example **class**, **loop** etc.
- Predefined types, entities or routines such as *INTEGER*, *Result* or *Create* appear in italic font and stand for themselves.
- Special symbols are enclosed in double quotes, for example ";", ":", etc. The double quote character is written in simple quotes as '"' (the simple quote character is written as "'").

Alternative right-hand sides are separated by vertical bars, as in

Type = *BOOLEAN* | *INTEGER* | *CHARACTER* | *REAL* |
Class_type | Association

where the first four alternatives are terminals and the last two are references to non-terminals defined elsewhere.

Two notational simplifications are used in right-hand sides:

- [comp] denotes the optional presence of an optional component comp;
- {Construct § ...} describes sequences of **zero or more** instances of Construct, separated from each other, if more than one, by the separator §.
- {Construct § ...}$^+$ describes sequences of **one or more** instances of Construct, separated from each other, if more than one, by the separator §.

Note that special symbols are quoted, so that there is no danger of confusion between the meta-symbols of this notation, such as [, {, + etc., and corresponding symbols in the language described, which will appear as "[", "{", "+" etc.

As an example of this notation, the following describes a trivial language with instructions "skip" and "goto", each instruction being possibly labeled, and separated from the next by a semicolon.

Warning: this is not the syntax of Eiffel!

Program	=	{Instruction ";" ...}
Instruction	=	[Label ":"] Simple_instruction
Simple_instruction	=	Skip | Goto
Skip	=	**skip**
Goto	=	**goto** Label
Label	=	Identifier

Contents