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 hitting the shores of the computing world.

"Object-oriented" is the latest *in* term, complementing and in many cases 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) "it cannot work"; (3) "that's how I did it all along anyway". (The order may vary.)

Let us have this clear right away, lest the reader think the author takes a half-hearted approach to his topic: I do not see the object-oriented method as 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 techniques that most people still use today — including some of the principles taught in many software engineering textbooks. I further believe that object technology holds the potential for fundamental changes in the software industry, 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 analysis, design and implementation.

"Avenue to software analysis, design and implementation". To present the object-oriented method, this books resolutely takes the viewpoint of software engineering — of the methods, tools and techniques for developing quality software in production environments. This is not the only possible perspective, as there has also been interest in applying object-oriented principles to such areas as exploratory programming and artificial intelligence. Although the presentation does not exclude these applications, they are not its main emphasis. Our principal goal in this discussion is to study how practicing software developers, in industrial as well as academic environments, can use object technology to improve (in some cases dramatically) the quality of the software they produce.

Structure, reliability, epistemology and classification

Object technology is at its core the combination of four ideas: a structuring method, a reliability discipline, an epistemological principle and a classification technique.

The *structuring method* applies to software decomposition and reuse. Software systems perform certain actions on objects of certain types; to obtain flexible and reusable systems, it is better to base their structure on the object types than on the actions. The resulting concept is a remarkably powerful and versatile mechanism called the **class**, which in object-oriented software construction serves as the basis for both the modular structure and the type system.

The *reliability discipline* is a radical approach to the problem of building software that does what it is supposed to do. The idea is to treat any system as a collection of components which collaborate the way successful businesses do: by adhering to **contracts** defining explicitly the obligations and benefits incumbent on each party.

The *epistemological principle* addresses the question of how we should describe the classes. In object technology, the objects described by a class are only defined by what we can do with them: operations (also known as *features*) and formal properties of these operations (the contracts). This idea is formally expressed by the theory of **abstract data types**, covered in detail in a chapter of this book. It has far-reaching implications, some going beyond software, and explains why we must not stop at the naïve concept of "object" borrowed from the ordinary meaning of that word. The tradition of information systems modeling usually assumes an "external reality" that predates any program using it; for the object-oriented developer, such a notion is meaningless, as the reality does not exist independently of what you want to do with it. (More precisely whether it exists or not is an irrelevant question, as we only know what we can use, and what we know of something is defined entirely by how we can use it.)

Abstract data types are discussed in chapter 6, which also addresses some of the related epistemological issues.

The *classification technique* follows from the observation that systematic intellectual work in general and scientific reasoning in particular require devising taxonomies for the domains being studied. Software is no exception, and the object-oriented method relies heavily on a classification discipline known as **inheritance**.

Simple but powerful

The four concepts of class, contract, abstract data type and inheritance immediately raise a number of questions. How do we find and describe classes? How should our programs manipulate classes and the corresponding objects (the *instances* of these classes)? What are the possible relations between classes? How can we capitalize on the commonalities that may exist between various classes? How do these ideas relate to such key software engineering concerns as extendibility, ease of use and efficiency?

Answers to these questions rely on a small but powerful array of techniques for producing reusable, extendible and reliable software: polymorphism and dynamic binding; a new view of types and type checking; genericity, constrained and

unconstrained; information hiding; assertions; safe exception handling; automatic garbage collection. Efficient implementation techniques have been developed which permit applying these ideas successfully to both small and large projects under the tight constraints of commercial software development. Object-oriented techniques have also had a considerable impact on user interfaces and development environments, making it possible to produce much better interactive systems than was possible before. All these important ideas will be studied in detail, so as to equip the reader with tools that are immediately applicable to a wide range of problems.

Organization of the text

In the pages that follow we will review the methods and techniques of object-oriented software construction. The presentation has been divided into six parts.

Chapters 1 to 2.

Part A is an introduction and overview. It starts by exploring the fundamental issue of software quality and continues with a brief survey of the method's main technical characteristics. This part is almost a little book by itself, providing a first view of the object-oriented approach for hurried readers.

Chapters 3 to 6.

Part B is not hurried. Entitled "The road to object orientation", it takes the time to describe the methodological concerns that lead to the central O-O concepts. Its focus is on modularity: what it takes to devise satisfactory structures for "in-the-large" system construction. It ends with a presentation of abstract data types, the mathematical basis for object technology. The mathematics involved is elementary, and less mathematically inclined readers may content themselves with the basic ideas, but the presentation provides the theoretical background that you will need for a full understanding of O-O principles and issues.

Chapters 7 to 18.

Part C is the technical core of the book. It presents, one by one, the central technical components of the method: classes; objects and the associated run-time model; memory management issues; genericity and typing; design by contract, assertions, exceptions; inheritance, the associated concepts of polymorphism and dynamic binding, and their many exciting applications.

Chapters 19 to 29.

Part D discusses methodology, with special emphasis on analysis and design. Through several in-depth case studies, it presents some fundamental *design patterns*, and covers such central questions as how to find the classes, how to use inheritance properly, and how to design reusable libraries. It starts with a meta-level discussion of the intellectual requirements on methodologists and other advice-givers; it concludes with a review of the software process (the lifecycle model) for O-O development and a discussion of how best to teach the method in both industry and universities.

Chapters 30 to 32.

Part E explores advanced topics: concurrency, distribution, client-server development and the Internet; persistence, schema evolution and object-oriented databases; the design of interactive systems with modern ("GUI") graphical interfaces.

Part F is a review of how the ideas can be implemented, or in some cases emulated, in various languages and environments. This includes in particular a discussion of major object-oriented languages, focusing on Simula, Smalltalk, Objective-C, C++, Ada 95 and Java, and an assessment of how to obtain some of the benefits of object orientation in such non-O-O languages as Fortran, Cobol, Pascal, C and Ada.

Chapters 33 to 35.

Part G (*doing it right*) describes an environment which goes beyond these solutions and provides an integrated set of tools to support the ideas of the book.

Chapter 36.

As complementary reference material, an appendix shows some important reusable library classes discussed in the text, providing a model for the design of reusable software.

Appendix A.

A Book-Wide Web

It can be amusing to see authors taking pains to describe recommended paths through their books, sometimes with the help of sophisticated traversal charts — as if readers ever paid any attention, and were not smart enough to map their own course. An author is permitted, however, to say in what spirit he has scheduled the different chapters, and what path he had in mind for what Umberto Eco calls the Model Reader — not to be confused with the real reader, also known as "you", made of flesh, blood and tastes.

The answer here is the simplest possible one. This book tells a story, and assumes that the Model Reader will follow that story from beginning to end, being however invited to avoid the more specialized sections marked as "skippable on first reading" and, if not mathematically inclined, to ignore a few mathematical developments also labeled explicitly. The real reader, of course, may want to discover in advance some of the plot's later developments, or to confine his attention to just a few subplots; every chapter has for that reason been made as self-contained as possible, so that you should be able to intake the material at the exact dosage which suits you best.

Because the story presents a coherent view of software development, its successive topics are tightly intertwined. The margin notes offer a subtext of cross references, a Book-Wide Web linking the various sections back and forth. My advice to the Model Reader is to ignore them on first reading, except as a reassurance that questions which at some stage are left partially open will be fully closed later on. The real reader, who may not want any advice, might use the cross references as unofficial guides when he feels like cheating on the prearranged order of topics.

Both the Model Reader and the real reader should find the cross references mostly useful in subsequent readings, to make sure that they have mastered a certain object-oriented concept in depth, and understood its connections with the method's other components. Like the hyperlinks of a WWW document, the cross references should make it possible to follow such associations quickly and effectively.

The CD-ROM that accompanies this book and contains all of its text provides a convenient way to follow cross references: just click on them. All the cross references have been preserved.

See "About the accompanying CD-ROM", page xiv.

The notation

In software perhaps even more than elsewhere, thought and language are closely connected. As we progress through these pages, we will carefully develop a notation for expressing object-oriented concepts at all levels: modeling, analysis, design, implementation, maintenance.

Here and everywhere else in this book, the pronoun "we" does not mean "the author": as in ordinary language, "we" means you and I — the reader and the author. In other words I would like you to expect that, as we develop the notation, you will be involved in the process.

This assumption is not really true, of course, since the notation existed before you started reading these pages. But it is not completely preposterous either, because I hope that as we explore the object-oriented method and carefully examine its implications the supporting notation will dawn on you with a kind of inevitability, so that you will indeed feel that you helped design it.

This explains why although the notation has been around for more than ten years and is in fact supported by several commercial implementations, including one from my company (ISE), I have downplayed it as a language. (Its name does appear in one place in the text, and several times in the bibliography.) This book is about the object-oriented method for reusing, analyzing, designing, implementing and maintaining software; the language is an important and I hope natural consequence of that method, not an aim in itself.

In addition, the language is straightforward and includes very little else than direct support for the method. First-year students using it have commented that it was "no language at all" — meaning that the notation is in one-to-one correspondence with the method: to learn one is to learn the other, and there is scant extra linguistic decoration on top of the concepts. The notation indeed shows few of the peculiarities (often stemming from historical circumstances, machine constraints or the requirement to be compatible with older formalisms) that characterize most of today's programming languages. Of course you may disagree with the choice of keywords (why ***do*** rather than ***begin*** or perhaps ***faire***?), or would like to add semicolon terminators after each instruction. (The syntax has been designed so as to make semicolons optional.) But these are side issues. What counts is the simplicity of the notation and how directly it maps to the concepts. If you understand object technology, you almost know it already.

Most software books take the language for granted, whether it is a programming language or a notation for analysis or design. Here the approach is different; involving the reader in the design means that one must not only explain the language but also justify it and discuss the alternatives. Most of the chapters of part C include a "discussion" section explaining the issues encountered during the design of the notation, and how they were resolved. I often wished, when reading descriptions of well-known languages, that the designers had told me not only what solutions they chose, but why they chose them, and what alternatives they rejected. The candid discussions included in this book should, I hope, provide you with insights not only about language design but also about software construction, as the two tasks are so strikingly similar.

Analysis, design and implementation

It is always risky to use a notation that externally looks like a programming language, as this may suggest that it only covers the implementation phase. This impression, however wrong, is hard to correct, so frequently have managers and developers been told that a gap of metaphysical proportions exists between the ether of analysis-design and the underworld of implementation.

Well-understood object technology reduces the gap considerably by emphasizing the essential unity of software development over the inevitable differences between levels of abstraction. This *seamless* approach to software construction is one of the important contributions of the method and is reflected by the language of this book, which is meant for analysis and design as well as for implementation.

"SEAMLESSNESS AND REVERSIBILITY", 28.6, page 930.

Unfortunately some of the recent evolution of the field goes against these principles, through two equally regrettable phenomena:

- Object-oriented implementation languages which are unfit for analysis, for design and in general for high-level reasoning.
- Object-oriented analysis or design methods which do not cover implementation (and are advertized as "language-independent" as if this were a badge of honor rather than an admission of failure).

Such approaches threaten to cancel much of the potential benefit of the approach. In contrast, both the method and the notation developed in this book are meant to be applicable throughout the software construction process. A number of chapters cover high-level design issues; one is devoted to analysis; others explore implementation techniques and the method's implications on performance.

The environment

Software construction relies on a basic tetralogy: method, language, tools, libraries. The method is at the center of this book; the language question has just been mentioned. Once in a while we will need to see what support they may require from tools and libraries. For obvious reasons of convenience, such discussions will occasionally refer to ISE's object-oriented environment, with its set of tools and associated libraries.

The environment is used only as an example of what can be done to make the concepts practically usable by software developers. Be sure to note that there are many other object-oriented environments available, both for the notation of this book and for other O-O analysis, design and implementation methods and notations; and that the descriptions given refer to the state of the environment at the time of writing, subject, as anything else in our industry, to change quickly — for the better. Other environments, O-O and non O-O, are also cited throughout the text.

The last chapter, 36, summarizes the environment.

Acknowledgments (quasi-absence thereof)

The first edition of this book contained an already long list of thanks. For a while I kept writing down the names of people who contributed comments or suggestions, and then at some stage I lost track. The roster of colleagues from whom I have had help or borrowed ideas has now grown so long that it would run over many pages, and would inevitably omit some important people. Better then offend everyone a little than offend a few very much.

A few notes in the margin or in chapter-end bibliographic sections give credit for some specific ideas, often unpublished.

So these acknowledgments will for the most part remain collective, which does not make my gratitude less deep. My colleagues at ISE and SOL have for years been a daily source of invaluable help. The users of our tools have generously provided us with their advice. The readers of the first edition provided thousands of suggestions for improvement. In the preparation of this new edition (I should really say of this new book) I have sent hundreds of e-mail messages asking for help of many different kinds: the clarification of a fine point, a bibliographical reference, a permission to quote, the details of an attribution, the origin of an idea, the specifics of a notation, the official address of a Web page; the answers have invariably been positive. As draft chapters were becoming ready they were circulated through various means, prompting many constructive comments (and here I must cite by name the referees commissioned by Prentice Hall, Paul Dubois, James McKim and Richard Wiener, who provided invaluable advice and corrections). In the past few years I have given countless seminars, lectures and courses about the topics of this book, and in every case I learned something from the audience. I enjoyed the wit of fellow panelists at conferences and benefited from their wisdom. Short sabbaticals at the University of Technology, Sydney and the Università degli Studi di Milano provided me with a influx of new ideas — and in the first case with three hundred first-year students on whom to validate some of my ideas about how software engineering should be taught.

The large bibliography shows clearly enough how the ideas and realizations of others have contributed to this book. Among the most important conscious influences are the Algol line of languages, with its emphasis on syntactic and semantic elegance; the seminal work on structured programming, in the serious (Dijkstra-Hoare-Parnas-Wirth-Mills-Gries) sense of the term, and systematic program construction; formal specification techniques, in particular the inexhaustible lessons of Jean-Raymond Abrial's original (late nineteen-seventies) version of the Z specification language, his more recent design of B, and Cliff Jones's work on VDM; the languages of the modular generation (in particular Ichbiah's Ada, Liskov's CLU, Shaw's Alphard, Bert's LPG and Wirth's Modula); and Simula 67, which introduced most of the concepts many years ago and had most of them right, bringing to mind Tony Hoare's comment about Algol 60: that it was such an improvement over most of its successors.

Foreword to the second edition

*M*any events have happened in the object-oriented world since the first edition of *OOSC* (as the book came to be known) was published in 1988. The explosion of interest alluded to in the Preface to the first edition, reproduced in the preceding pages in a slightly expanded form, was nothing then as compared to what we have seen since. Many journals and conferences now cover object technology; Prentice Hall has an entire book series devoted to the subject; breakthroughs have occurred in such areas as user interfaces, concurrency and databases; entire new topics have emerged, such as O-O analysis and formal specification; distributed computing, once a specialized topic, is becoming relevant to more and more developments, thanks in part to the growth of the Internet; and the Web is affecting everyone's daily work.

This is not the only exciting news. It is gratifying to see how much progress is occurring in the software field — thanks in part to the incomplete but undeniable spread of object technology. Too many books and articles on software engineering still start with the obligatory lament about the "software crisis" and the pitiful state of our industry as compared to *true* engineering disciplines (which, as we all know, never mess things up). There is no reason for such doom. Oh, we still have a long, long way to go, as anyone who uses software products knows all too well. But given the challenges that we face we have no reason to be ashamed of ourselves as a profession; and we are getting better all the time. It is the ambition of this book, as it was of its predecessor, to help in this process.

This second edition is not an update but the result of a thorough reworking. Not a paragraph of the original version has been left untouched. (Hardly a single line, actually.) Countless new topics have been added, including a whole chapter on concurrency, distribution, client-server computing and Internet programming; another on persistence and databases; one on user interfaces; one on the software lifecycle; many design patterns and implementation techniques; an in-depth exploration of a methodological issue on which little is available in the literature, how to use inheritance well and avoid misusing it; discussions of many other topics of object-oriented methodology; an extensive presentation of the theory of abstract data types — the mathematical basis for our subject, indispensable to a complete understanding of object technology yet seldom covered in detail by textbooks and tutorials; a presentation of O-O analysis; hundreds of new bibliographic and Web site references; the description of a complete object-oriented development environment (also included on the accompanying CD-ROM for the reader's enjoyment) and of the underlying concepts; and scores of new ideas, principles, caveats, explanations, figures, examples, comparisons, citations, classes, routines.

The reactions to *OOSC*-1 have been so rewarding that I know readers have high expectations. I hope they will find *OOSC*-2 challenging, useful, and up to their standards.

Santa Barbara — B.M.
January 1997

About the accompanying CD-ROM

The CD-ROM that comes with this book contains the **entire hyperlinked text** in Adobe Acrobat format. It also includes Adobe's Acrobat Reader software, enabling you to read that format; the versions provided cover major industry platforms. If you do not already have Acrobat Reader on your computer, you can install it by following the instructions. The author and the publisher make no representations as to any property of Acrobat and associated tools; the Acrobat Reader is simply provided as a service to readers of this book, and any Acrobat questions should be directed to Adobe. You may also check with Adobe about any versions of the Reader that may have appeared after the book.

To get started with the CD-ROM, open the Acrobat file *README.pdf* in the OOSC-2 directory, which will direct you to the table of contents and the index. You can only open that file under Acrobat Reader; if the Reader has not been installed on your computer, examine instead the plain-text version in the file *readme.txt* in the top-level directory.

The presence of an electronic version will be particularly useful to readers who want to take advantage of the thousands of cross-references present in this book (see "A Book-Wide Web", page viii). Although for a first sequential reading you will probably prefer to follow the paper version, having the electronic form available on a computer next to the book alllows you to follow a link once in a while without having to turn pages back and forth. The electronic form is particularly convenient for a later reading during which you may wish to explore links more systematically.

All links (cross-references) appear in blue in the Acrobat form, as illustrated twice above (but not visible in the printed version). To follow a link, just click on the blue part. If the reference is to another chapter, the chapter will appear in a new window. The Acrobat Reader command to come back to the previous position is normally Control-minus-sign (that is, type – while holding down the CONTROL key). Consult the on-line Acrobat Reader documentation for other useful navigational commands.

Bibliographical references also appear as links, such as [Knuth 1968], in the Acrobat form, so that you can click on any of them to see the corresponding entry in the bibliography of appendix E.

The CD-ROM also contains:

- Library components providing extensive material for Appendix A.
- A chapter from the manual for a graphical application builder, providing mathematical complements to the material of chapter 32.

In addition, the CD-ROM includes a time-limited version of an advanced **object-oriented development environment** for Windows 95 or Windows NT, as described in chapter 36, providing an excellent hands-on opportunity to try out the ideas developed throughout the book. The "Readme" file directs you to the installation instructions and system requirements.

Acknowledgments: The preparation of the hyperlinked text was made possible by the help of several people at Adobe Inc., in particular Sandra Knox, Sarah Rosenbaum and the FrameMaker Customer Support Group.

On the bibliography, Internet sources and exercises

This book relies on earlier contributions by many authors. To facilitate reading, the discussion of sources appears in most cases not in the course of the discussion, but in the "Bibliographical notes" sections at chapter end. Make sure you read these sections, so as to understand the origin of many ideas and results and find out where to learn more.

The bibliography starts on page 1203.

References are of the form [*Name* 19xx], where *Name* is the name of the first author, and refer to the bibliography in appendix E. This convention is for readability only and is not intended to underrate the role of authors other than the first. The letter M in lieu of a *Name* denotes publications by the author of this book, listed separately in the second part of the bibliography.

Aside from the bibliography proper, some references appear in the margin, next to the paragraphs which cite them. The reason for this separate treatment is to make the bibliography usable by itself, as a collection of important references on object technology and related topics. Appearance as a margin reference rather than in the bibliography does not imply any unfavorable judgment of value; the division is simply a pragmatic assessment of what belongs in a core list of object-oriented references.

***

Although electronic references will undoubtedly be considered a matter of course a few years from now, this must be one of the first technical books (other than books devoted to Internet-related topics) to make extensive use of references to World-Wide-Web pages, Usenet newsgroups and other Internet resources.

Electronic addresses are notoriously volatile. I have tried to obtain from the authors of the quoted sources some reassurance that the addresses given would remain valid for several years. Neither they nor I, of course, can provide an absolute guarantee. In case of difficulty, note that on the Net more things move than disappear: keyword-based search tools can help.

***

Most chapters include exercises of various degrees of difficulty. I have refrained from providing solutions, although many exercises do contain fairly precise hints. Some readers may regret the absence of full solutions; I hope, however, that they will appreciate the three reasons that led to this decision: the fear of spoiling the reader's enjoyment; the realization that many exercises are design problems, for which there is more than one good answer; and the desire to provide a source of ready-made problems to instructors using this book as a text.

***

For brevity and simplicity, the text follows the imperfect but long-established tradition of using words such as “he” and “his”, in reference to unspecified persons, as shortcuts for “he or she” and “his or her”, with no intended connotation of gender.

A modest soul is shocked by objects of such kind

And all the nasty thoughts that they bring to one's mind.

Molière, *Tartuffe,* Act III.