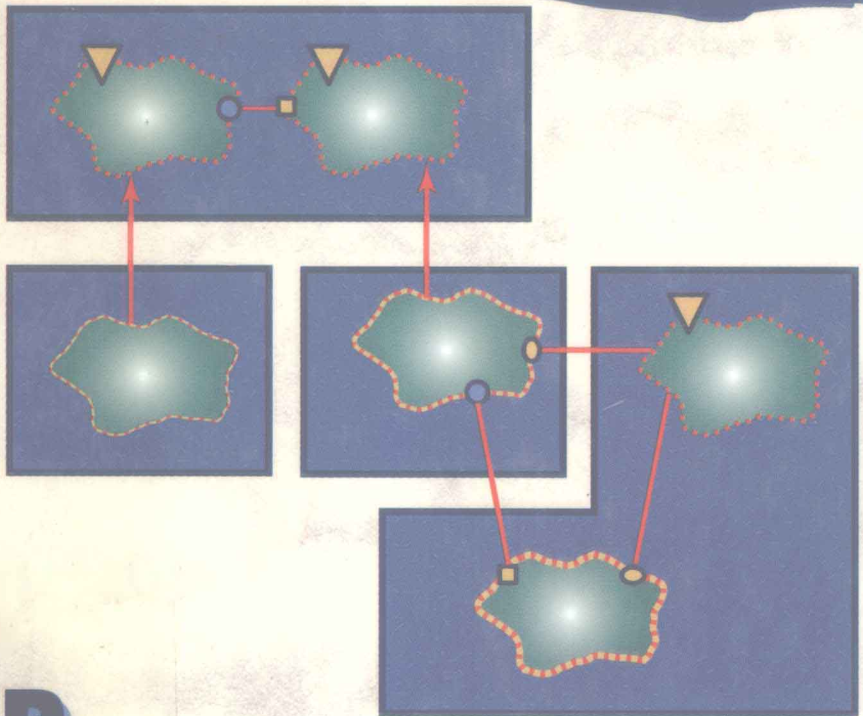
FOREWORD BY GRADY BOOCH

```
#include "PayrollDomain/namespaces.h"
#include "Components/namespaces.h"
using namespace Components;

#include "Components/string.h"
#include "Components/u_set.h"
//
// Name
//  Employee

class PayrollDomain::Employee
{
 public:
  void PayDay(const Date&);
  void SetClassification(PaymentClassifica
  void SetMethod(PaymentMethod*);
  void SetSchedule(PaymentSchedule*);

 private:
  double CalculatePay(const Date&) const;

  String              itsName;
  String              itsAddress;
  UnboundedSet<Affiliation*> itsAffiliatio
  PaymentSchedule*          itsSchedule;
  PaymentClassification*    itsClassificatio
  PaymentMethod*            itsMethod;
};
```

DESIGNING OBJECT-ORIENTED C++ APPLICATIONS USING THE BOOCH METHOD



ROBERT C. MARTIN

Designing Object-Oriented C++ Applications

Using the Booch Method

Robert Cecil Martin

Object Mentor Associates

Prentice Hall, Englewood Cliffs, New Jersey 07632

Martin, Robert C.
Designing object-oriented C++ applications using the Booch method / Robert C. Martin.
p. cm.
"An Alan R. Apt Book."
Includes index.
ISBN 0-13-203837-4
1. Object-oriented programming. 2. C++ (Computer program language) 3. Computer software--Development. I. Title.
QA76.64.M384 1995
005.13'3--dc20

94-47129
CIP

Publisher: Alan Apt
Production Editor: Mona Pompili
Cover Designer: Wendy Alling Judy
Copy Editor: Nick Murray
Production Coordinator: Lori Bulwin
Editorial Assistant: Shirley McGuire

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-203837-4

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*

For Ann Marie, Angela, Micah, Gina and Justin . . .

There is no greater treasure,
Nor any wealthier trove,
Than the company of my family,
And the comfort of their love.

Forward

By Grady Booch

Writing good software is indeed hard work. Moreover, the demand for quality software continues to grow at an insane pace, fueled by the increased connectivity of distributed computing systems and by greater user expectations for better visualization of and access to information. The good news is that this makes for very interesting times for the professional software developer.

Another piece of good news is that, over the past decade or so, developments in abstract data type theory, modularity, information modeling, and software process have evolved to provide the professional developer a sound collection of practices that can be used to attack this growing complexity. In many ways, these practices all come together in the form of object-oriented technology. Most notably, a whole family of object-oriented programming languages, such as C++ and Smalltalk, have emerged. In addition, a variety of object-oriented analysis and design methods have been proposed that exploit these object-oriented languages and help us model complex systems.

Yet, for the developer building real systems under very real schedules and limited resources, all theory is irrelevant if it is not pragmatic. Reality is that languages such as C++ and Smalltalk are not simple, and the applications to which we direct them are even less simple. Just because you are using an object-oriented programming language does not mean that all your projects will automatically be on schedule, under budget, and free of all flaws.

Robert Martin is, first and foremost, a very pragmatic developer. I've had the honor to work with him, and I've learned many things from his experience. This book speaks clearly of that experience. In this work, you will understand how to apply C++ effectively. You will also learn how to apply C++ in the context of the Booch method of object-oriented analysis and design, a method that provides a unification of what we know as the best practices today in object-oriented development.

Enjoy. I know I did in reading Robert's manuscript, and I'm sure you'll gain many useful — and ultimately very practical — insights as well.

Grady Booch
Chief Scientist
Rational Software Corporation

Preface

Software design is hard, and we need all the help we can get.

— Bjarne Stroustrup, 1991

Software design *is* hard. We are now well into the fifth decade since John Von Neumann conceived of the notion of a stored program. And although there have been many advances in both the theory and practice of software engineering, in comparison to our need, those advances have been precious and few. Object-Oriented Design is one of those advances and is the subject of this book.

Let's get one thing straight. Object-Oriented Design is not going to save the software world—it's not even going to come close. Applications designed using OOD will still be difficult to estimate, will still be difficult to implement, will still be difficult to maintain; and will still have bugs. Software design will still be hard. What OOD *will* do is provide some useful new tools that you can employ while designing software applications. Those tools, properly employed, will help you to manage the complexity of designing, implementing and supporting a software design. They will also help you to build your designs in terms of reusable high level components.

OOD is a complex topic. Many books have been written on the topic. Most of them describe OOD in terms of its definitions, notations, and methods. This book looks at the subject from the point of view of its *practice*. It tries to answer the question: "How do I do OOD?" In trying to answer that question I will employ two specific tools: *The Booch Notation*[1] for recording object-oriented designs and the C++[2] programming language, a language that supports object-oriented programming. We will explore how designs can be documented and manipulated using Booch's notation, and how they can be translated into C++. During this exploration we will encounter the important principles of OOD and investigate many ways in which they may be employed.

1. Grady Booch, *Object-Oriented Design with Applications*, (Copyright © 1991 by The Benjamin/Cummings Publishing Company).
2. Bjarne Stroustrup, *The C++ Programming Language, 2d. ed.* (Addison Wesley, 1991).

About This Book

Goals/Purpose

There are many books describing the various practices of object-oriented Design (OOD). There are many other books describing the syntax and usage of C++. This book is a synthesis of these two concepts. C++ is a rich and expressive language. Having C as a subset may encourage software engineers to use it as "a better C." While this is not altogether a bad thing, it falls short of the potential benefits that a true object-oriented approach could yield. This book presents the fundamental concepts of object-oriented design and shows how to apply those concepts using C++. The approach is a practical, problem-solving presentation, written for those who are, or aspire to become, practitioners of object-oriented design. Special attention is given to traps, pitfalls, and techniques in the application of C++ to OOD.

The Booch notation was chosen as the representational vehicle for OOD because of its popularity, scalability, and notational elegance. The notation is explored in detail, and is used to present the concepts of OOD. Where appropriate, the notation is translated into corresponding C++ code. This provides the reader with a "Rosetta stone" describing the linkage between the abstract OOD notation and the syntax of C++.

The practices of software engineering receive special attention, both in the creation of the logical design, and the physical development environment. The methods for designing and developing "big" software are discussed in detail. The goal is to provide the tools needed to deal with large and complex projects.

Audience

This book is for software engineers—specifically for those who are interested in learning how to design applications using object-oriented design techniques, and who want to implement those applications in C++. It is assumed that the reader has a minimal working knowledge of C++.

You should be prepared to work hard while reading this book. Quite a bit of detail is presented, and you will benefit by studying it carefully. You can also browse the book to gain a general notion of OOD, its representation in the Booch notation, and its ultimate expression in C++. However, diligence in learning these techniques, as opposed to just skimming them, will be well worth the effort.

Anatomy and Physiology of Design

OOD is a complex discipline. It has its own vernacular, full of words, diagrams, principles, and concepts. No reasonable discussion about OOD can occur until you have learned this vocabulary. Thus the first five chapters of this book are a step-by-step anatomy and physiology of OOD. They present the concepts, definitions and principles of OOD by exploring a number of relatively simple case studies. These case studies come from a variety of application domains, so that you can learn how to use OOD and C++ to solve diverse sets of problems. Each case study works through a simple object-oriented design, and often shows its implementation in C++.

The first five chapters also present the Booch notation as a vehicle for recording and manipulating object-oriented design decisions. The Booch notation is "large"; it provides many notational conventions for dealing with issues at all levels of the design. Each of the first five chapters explores a different part of the notation and how it applies to OOD and C++.

The next three chapters demonstrate how to apply the practices and principles of OOD to a problem of significant size. They are an expedition through the analysis, high-level design, low-level design, and physical design of a single complex application.

During this expedition, we thoroughly discuss the methods and rationale behind the important design decisions. Also, many false starts and design errors are documented. These errors are real, not contrived examples; I actually made these design errors while writing this book. Furthermore the methods by which these errors are discovered and solved are represented as faithfully as possible.

By reading through these chapters, you will follow the path that I took while doing the design, including the dead ends, cul-de-sacs, and wrong turns. In my opinion, studying design errors is just as educational (if not more educational) as studying "correct" solutions.

Software Is Hard

The real crux of the "software crisis" (and the real reason why books like this are necessary) is that software *is* hard. An application comprises myriads of intricate little details. It is hard to weave all those details into a working program. Why should this be so? Why is a concept as easy to grasp as a word processor, for example, so hideously complex to implement? It is because humans are so good at abstracting away details.

By using the two words "word processor" I have described a broad class of highly complex and intricate applications. I have also abstracted away all the details involved with those applications. Software is hard because we are so good at envisioning abstract applications without thinking about the details. Somewhere in our gut we *know* what we want an application to do. We don't have to describe it in detail, we just *know*. We *know* what a word processor does. It's obvious. The concept is simple, and so we expect the task of writing the software to be simple. Only when we enumerate the enormous amount of details involved with making a real word processor do we begin to get a feeling for the true complexity of the application. However, enumeration is not enough. Those details must be organized, orchestrated, and intermixed with the details and limitations of the language and machine. The result is a daunting task requiring far more time and effort than generally expected. Software is hard because it takes an enormous effort to produce applications that can be so simply conceived of. Software is hard because it takes so much effort to keep pace with users' requirements and expectations. Software is hard because it is so easy to dream up.

On the other hand, computer hardware is far outstripping the lay person's expectations. It used to be that a computer would require hundreds of engineers to design and build. Nowadays, a moderately skilled hardware engineer can tinker a far superior computer together in the basement. Computer memory used to be hideously expensive, large, and power-hungry. Now we buy gigabyte disks the size of pocket radios, and put them in the multimegabyte, multimegahertz computers that we keep in our briefcases.

This creates a double-whammy for the software crisis. Not only are users' expectations growing faster than our ability to produce workable applications, but the computers themselves are improving faster still. Users reasonably expect that more powerful computers should have more powerful applications to run on them. Unfortunately the power of the computer doesn't provide much assistance in managing the complexity of a huge software project.

To be sure, there have been some very powerful improvements in software technology. But it is nothing near the sort of wild expansion that computer hardware has seen. Suppose that two software engineers are separately trying to write the same program. One is using 1990s software-development technology, and the other is using the tools and techniques from the 1950s. Given that the application can be handled by the corresponding hardware technologies, how much more efficient would the modern engineer be? Is she 30% more efficient? Twice as efficient? Could she be ten times as efficient? Whatever the answer, it will in no way compare with the sheer orders of magnitude by which the efficiency of hardware engineers has increased. One hardware engineer in the 1990s can implement what took armies of engineers to implement in the 1950s.

Hardware engineers are so much more efficient today because they have developed a technology that allows them to build upon each other's work. Nobody has to redesign a flip-flop.[3] Nobody has to build a flip-flop. You can buy them by the thousands in little integrated circuits. Engineers don't have to design op-amps, or adders, or CPUs, or any of the other staples of electronic devices. They can just buy the building blocks and tie them together using standard electronic engineering techniques. And every year the building blocks become more powerful and more complex. Every year this encapsulated power and complexity is made directly available for hardware engineers to bring to bear upon their

3. An electronic memory device capable of storing 1 bit of information.

designs. It is this encapsulation and mass production of complexity that has so magnified the power of the hardware engineer.

No such revolution has yet taken place in the realm of software engineering. Oh, we might not have to write **`sort`** functions anymore.[4] And maybe most of our I/O is taken care of for us by an operating system. But nobody is out there selling "Integrated Application Modules" that software engineers can buy and hook together with the kind of efficiency experienced by hardware engineers. As a group, software engineers are not building upon each other's work. We continue to reimplement different variations of the same functions over and over again.

OOD Can Make Software "Softer"

Software may be hard, but OOD can help to soften it a bit. OOD provides the tools and techniques by which we can encapsulate a certain level of functionality and complexity. Using OOD, we can create black-box software modules that hide a great deal of complexity behind a simple interface. Software engineers can tie these black boxes together using standard software techniques.

Booch calls these black boxes *class categories*. Class categories are comprised of entities known as *classes*, which are bound together by *class relationships*. A great deal of complexity can be buried in a class, while its interfaces can remain relatively simple. This takes advantage of the fact that people are so good at abstracting away details. If we can bury all the details away in some class, then we don't have to think about them any more. We can use the class as often as we like, and for as many applications as we see fit, but we never have to consider the details buried inside it again. This is an enormously powerful concept. Such classes allow us to wield great power at relatively low cost.

Class categories organize groups of classes into mechanisms that implement high-level policies, *independent of the details that they control.* Such categories, designed for one application, can be reused in many other applications that require the same kind of policies.

By 1990 there were some class libraries for sale. These libraries provided basic, low-level classes such as queues, linked-lists, sorted-lists, complex numbers, and so on. They provided abstractions that software engineers could readily build upon and incorporate into their own class categories. By 1992 there were some libraries of class categories for sale that encapsulated the policies that managed particular graphic user interfaces. These category libraries, also called "frameworks", made it much simpler to design and implement applications that employed those GUIs. Again, these frameworks can be built upon to ease the job of designing and building software.

Will more libraries of class categories appear on the market, with more and more policies and functionality and complexity buried within simple interfaces? Will it one day be possible to buy a "Word Processor" framework, or a "Spreadsheet" framework? Will soft-

4. Although I'd be willing to bet that many of you have within the last year or so.

ware engineers be able to tie such wildly complex entities together with the same ease and efficiency that hardware engineers currently tie CPUs, UARTs, and RAMs together? That would certainly be a worthy goal. It may be that OOD can move us closer to achieving it.

Acknowledgements

Question: How can I get you to read this section? Without the acknowledged individuals, this work would not have been possible. These people deserve to be recognized, and so I want you to read their names. But you won't do that if I simply list their names, so I am going to give you some incentive to get to know these people. Please read on.

I first became interested in object-oriented programming (OOP) in 1985. I read Adele Goldberg's excellent books on Smalltalk-80, and bought a Smalltalk compiler for my Macintosh, which taught me much about OOP. At the same time, I attempted to implement pseudo object-oriented inheritance and message-dispatching mechanisms in C. This turned out to be very difficult, and I eventually abandoned the idea, but not before I had several stimulating conversations with a friend and associate named Jim Newkirk. We spent many hours discussing object-oriented design while attempting to use the "inheritance mechanisms" that I had invented for C.

In 1986 I got a copy of Bjarne Stroustrup's first edition of *The C++ Programming Language*. Its similarity in size and style to Kernighan and Ritchie's wonderful *The C Programming Language* was extremely compelling. I said to myself, "Oh, this must be the next C."

After reading this book, and becoming enthralled by the language, I found myself leading a one-man abortive campaign to get my employer to purchase a C++ compiler. But the cost, at that time, was high, and the interest among my co-workers was low, so I was unable to achieve this goal. It was difficult to convince people even that strong typing would be beneficial, let alone attempt to explain the odd appellation of object-oriented programming.

It was not until 1989 that I was able to get my hands on a real C++ compiler. By that time I had written thousands of lines of object-oriented code, but none in C++. I bought a copy of Dewhurst and Stark's *Programming in C++* and I quickly began exploring the language and getting used to its quirks and features. In this process, I am not even nearing completion.

In 1990 I read Coad's *OOA*, Booch's *Object-Oriented Design with Applications*, and *Designing Object-Oriented Software* by Wirfs-Brock, Wilkerson, and Wiener. I began applying the techniques of OOD that I learned in these books to the applications I was writing for my employer. I also adopted Booch's notation as my prime mechanism for recording design decisions. I was pleasantly surprised that the notation, odd as it may seem at first glance, was actually quite easy to draw by hand and allowed me to record my designs with a density that I had not experienced with other notational methods.

In the middle of 1991, my co-worker, Bill Vogel, got a call from a recruiter looking for object-oriented engineers. After some discussion with the agent on the other end of the phone, he handed the phone over to me and said: "Uncle Bob, I think this is for you." That phone call eventually led me to leave my employer and begin working as a consultant for Rational, where Grady Booch was employed as chief scientist.

At Rational I worked on a product called "Rose." I was fortunate enough to work with Grady on several projects during my months there. I was also fortunate to be working with a group of some of the most astounding engineers that I have ever met. The Rose team was awesome in the sheer brain-power of its members. I learned a great deal from all of them. There were, to mention a few, Paul Rogers, the cool-head; Bob Weissman, the trouble-shooter; Paul Jasper, the sound-man; Dave Stevenson, the rocket scientist; and Mike Higgs, the pragmatist.

While working at Rational, I conceived the idea for this book. I wrote drafts of the first three chapters, and showed them to Grady. He was kind enough to read and review them, and then to guide me into the publishing process.

Grady introduced me to Alan Apt of Prentice Hall, who is the editor of this book. Alan is a diligent and enthusiastic editor. He recruited some of the industry's top people to review my work—names like Steve Buroff, Jim Coplien, Mike Vilot, and Stan Lippman. Of Stan's reviews I will say only that they caused me, at once, the most pain and profit.

Of course this was just the kick-off. Many other people reviewed my work. Brett Schuchert, Bob Weissman, and Mike Higgs deserve special mention. To Jim Newkirk, who became the sounding-board for many of my partially formed ideas, and who made many contributions of his own, I would like to say a heartfelt thank you.

Writing a book takes a great deal of time, and for a man who is supporting a large family, that time must be taken away from that family. This book could never even have been conceived, let alone written, if not for the faithful and loving support provided to me by my wonderful wife Ann Marie, and all my terrific kids: Angela, Micah, Gina, and Justin. Every man should be as lucky as I am.

The production of this book was an arduous task, and would have been beyond my abilities were it not for the unfailing efforts of Jennifer Kohnke and Mona Pompili, Jim Newkirk and Bhama Rao. The harder I worked, the harder they worked. Thanks.

Much of what appears in the pages to come has been significantly influenced by the fluid and incredibly dynamic discussions that appear on the net. Among those "netters" who have had a profound influence on my thinking are Jim Adcock, Steve Clamage, Jamshid Afshar, Mark Terribile, John Skaller, Scott Meyers, Marshall Cline, Paul Lucas, Red Mitchell, and John Goodsen.

Finally, there are several people who deserve mention because they have influenced my thinking concerning software over the last two decades. Of course if they were all enumerated, the list would fill many dozens of pages. So I will constrain myself to mention just these few: Dave Lasker, who helped a would-be consultant achieve his goals; Ken Finder, who taught wisdom to a fool; Jerry Fitzpatrick, who believed the weakling could defeat the giant; and Tim Conrad, who turned impossible dreams into a few week's joyous labor.

Contents