UML与面向对象设计影印丛书

面向对象系统架构及设计

THE ART OF OBJECTS

OBJECTS-ORIENTED DESIGN AND ARCHITECTURE

YUN-TUNG LAU,PH.D. 编著

科学出版社
www.sciencep.com

UML与面向对象设计影印丛书

面向对象系统架构及设计

Yun-Tung Lau，Ph.D. 编著

科学出版社

北京

内 容 简 介

本书针对面向对象开发过程中的系统设计阶段，全面论述了基本概念、静态设计模式、对象一致性、对象建模中的高级问题、动态对象建模、常用接口、面向对象系统架构等重要内容。书中给出了大量的实例，并提供了网站以便读者下载。

本书适合面向对象系统分析及设计人员阅读。

ISBN 0-201-71161-3

图字：01-2003-2539

图书在版编目(CIP)数据

面向对象系统架构及设计=The Art of Objects:Object Oriented Design and Architecture/（美）劳德（Laud,Y.）著.—影印本，—北京：科学出版社，2003

ISBN 7-03-011400-0

Ⅰ.面… Ⅱ.劳… Ⅲ.面向对象语言—程序设计—英文 Ⅳ.TP312

中国版本图书馆 CIP 数据核字（2003）第 030825 号

策划编辑：李佩乾/责任编辑：李佩乾
责任印制：吕春珉/封面制作：东方人华平面设计室

科学出版社 出版
北京东黄城根北街16号
邮政编码:100717
http://www.sciencep.com

双青印刷厂 印刷

科学出版社发行 各地新华书店经销

*

2003 年 5 月第 一 版 开本：787×960 1/16
2003 年 5 月第一次印刷 印张：23 3/4
印数：1—2 000 字数：451 000

定价：40.00 元

（如有印装质量问题，我社负责调换<环伟>）

影印前言

随着计算机硬件性能的迅速提高和价格的持续下降，其应用范围也在不断扩大。交给计算机解决的问题也越来越难，越来越复杂。这就使得计算机软件变得越来越复杂和庞大。20 世纪 60 年代的软件危机使人们清醒地认识到按照工程化的方法组织软件开发的必要性。于是软件开发方法从 60 年代毫无工程性可言的手工作坊式开发，过渡到 70 年代结构化的分析设计方法、80 年代初的实体关系开发方法，直到面向对象的开发方法。

面向对象的软件开发方法是在结构化开发范型和实体关系开发范型的基础上发展而来的，它运用分类、封装、继承、消息等人类自然的思维机制，允许软件开发者处理更为复杂的问题域和其支持技术，在很大程度上缓解了软件危机。面向对象技术发端于程序设计语言，以后又向软件开发的早期阶段延伸，形成了面向对象的分析和设计。

20 世纪 80 年代末 90 年代初，先后出现了几十种面向对象的分析设计方法。其中，Booch, Coad/Yourdon、OMT 和 Jacobson 等方法得到了面向对象软件开发界的广泛认可。各种方法对许多面向对象的概念的理解不尽相同，即便概念相同，各自技术上的表示法也不同。通过 90 年代不同方法流派之间的争论，人们逐渐认识到不同的方法既有其容易解决的问题，又有其不容易解决的问题，彼此之间需要进行融合和借鉴；并且各种方法的表示也有很大的差异，不利于进一步的交流与协作。在这种情况下，统一建模语言(UML)于 90 年代中期应运而生。

UML 的产生离不开三位面向对象的方法论专家 G. Booch、J. Rumbaugh 和 I. Jacobson 的通力合作。他们从多种方法中吸收了大量有用的建模概念，使 UML 的概念和表示法在规模上超过了以往任何一种方法，并且提供了允许用户对语言做进一步扩展的机制。UML 使不同厂商开发的系统模型能够基于共同的概念，使用相同的表示法，呈现彼此一致的模型风格。1997 年 11 月 UML 被 OMG 组织正式采纳为标准的建模语言，并在随后的几年中迅速地发展为事实上的建模语言国际标准。

UML 在语法和语义的定义方面也做了大量的工作。以往各种关于面向对象方法的著作通常是以比较简单的方式定义其建模概念，而以主要篇幅给出过程指导，论述如何运用这些概念来进行开发。UML 则以一种建模语言的姿态出现，使用语言学中的一些技术来定义。尽管真正从语言学的角度看它还有许多缺陷，但它在这方面所做的努力却是以往的各种建模方法无法比拟的。

从 UML 的早期版本开始，便受到了计算机产业界的重视，OMG 的采纳和大公司的支持把它推上了实际上的工业标准的地位，使它拥有越来越多的用户。它被广泛地用

于应用领域和多种类型的系统建模，如管理信息系统、通信与控制系统、嵌入式实时系统、分布式系统、系统软件等。近几年还被运用于软件再工程、质量管理、过程管理、配置管理等方面。而且它的应用不仅仅限于计算机软件，还可用于非软件系统，例如硬件设计、业务处理流程、企业或事业单位的结构与行为建模，等等。

在 UML 陆续发布的几个版本中，逐步修正了前一个版本中的缺陷和错误。即将发布的 UML2.0 版本将是对 UML 的又一次重大的改进。将来的 UML 将向着语言家族化、可执行化、精确化等理念迈进，为软件产业的工程化提供更有力的支撑。

本丛书收录了与面向对象技术和 UML 有关的 12 本书，反映了面向对象技术最新的发展趋势以及 UML 的新的研究动态。其中涉及对面向对象建模理论研究与实践的有这样几本书：《面向对象系统架构及设计》主要讨论了面向对象的基本概念、静态设计、永久对象、动态设计、设计模式以及体系结构等近几年来面向对象技术领域中的新的理论知识与方法；《用 UML 进行用况对象建模》主要介绍了面向对象的需求阶段、分析阶段、设计阶段中用况模型的建立方法与技术；《高级用况建模》介绍了在建立用况模型中需要注意的高级的问题与技术；《UML 面向对象设计基础》则侧重于经典的面向对象理论知识的阐述。

涉及 UML 在特定领域的运用的有这样几本：《UML 实时系统开发》讨论了进行实时系统开发时需要对 UML 进行扩展的技术；《用 UML 构建 Web 应用程序》讨论了运用 UML 进行 Web 应用建模所应该注意的技术与方法；《面向对象系统测试：模型、视图与工具》介绍了将 UML 应用于面向对象的测试领域所应掌握的方法与工具；《对象、构件、框架与 UML 应用》讨论了如何运用 UML 对面向对象的新技术——构件-框架技术建模的方法策略。《UML 与 Visual Basic 应用程序开发》主要讨论了从 UML 模型到 Visual Basic 程序的建模与映射方法。

介绍面向对象编程技术的有两本书：《COM 高手心经》和《ATL 技术内幕》，深入探讨了面向对象的编程新技术——COM 和 ATL 技术的使用技巧与技术内幕。

还有一本《Executable UML 技术内幕》，这本书介绍了可执行 UML 的理念与其支持技术，使得模型的验证与模拟以及代码的自动生成成为可能，也代表着将来软件开发的一种新的模式。

总之，这套书所涉及的内容包含了对软件生命周期的全过程建模的方法与技术，同时也对近年来的热点领域建模技术、新型编程技术作了深入的介绍，有些内容已经涉及到了前沿领域。可以说，每一本都很经典。

有鉴于此，特向软件领域中不同程度的读者推荐这套书，供大家阅读、学习和研究。

北京大学计算机系　蒋严冰 博士

Preface

Object-oriented programming relies on programming languages. However, the concepts of objects transcend any specific programming languages. Many design patterns offer efficient modeling of static and dynamic object relationships. They can be used as the building blocks for sophisticated software systems. Similarly, at a system level, object-oriented architecture provides a lucid, high-level description of interconnected objects.

Tools may change. Programming languages may go out of favor. Yet the foundation of object design and architecture, and the art of applying it, will remain sound for a long time.

This book systematically presents the basic concepts of objects and practical object design patterns (both static and dynamic). It helps readers to gain a deep understanding of the patterns, allowing them to find design solutions quickly. In addition, the topics are forward looking, encompassing persistent objects, distributed objects, interface design patterns, XML (eXtensible Markup Language) object models, Web applications with thin clients, and so forth. Going beyond the design level, the book discusses object-oriented architecture, covering clients/servers, multi-tier systems, federations, agents, and others.

The Unified Modeling Language (UML), especially its graphic notation, is used as the primary means of presentation. The contents are independent of specific programming languages, making the book a general-purpose reference. However, many exercises do relate to certain languages (mostly Java). They help bring the readers closer to implementation and foster a concrete understanding of the underlying concepts. In addition, a wide range of real-world case studies and examples help elucidate these concepts and their practical application.

I did not use UML to specify all the details of an object design. For example, the UML Object Constraint Language is not used. In my opinion, source code with adequate inline comments is the best place to document the detailed logic of object behaviors.

This book can be used as a textbook for university or industrial training courses, or as a reference book for courses on object-oriented programming languages. This book is also suitable as a reference for mid- to advanced-level software professionals and graduate students. Many exercises are derived from actual projects. They expose readers to the full complexity of real-world systems.

Organization of the Book

This book has nine chapters, including several integrated case studies throughout the book. Chapter 1 describes the basic concepts in object-oriented programming, which include object, class, association, aggregation, servant class, and inheritance. It also introduces some basic notations of UML.

Chapter 2 discusses the common patterns in static design. The focus here is on the static relationships between classes. The dynamic or time-dependent behaviors are left to later chapters. I systematically present simple and complex patterns. They allow object designers to design with patterns rather than with individual classes.

I note that the distinction between analysis and design is vague. Analysis is more on understanding the concepts in an application domain and investigating the requirements. Design is more on finding a solution and verifying that the solution fits the requirements. With a concrete understanding of the object concepts and the relationships behind the static patterns, one would naturally apply object analyses and designs in an iterative fashion. The ultimate criterion for an appropriate object design is its fitness to the requirements.

In Chapter 3 I first present the basic concepts on database management and persistent objects. I then discuss different strategies to make objects persistent, particularly those involving object-oriented databases and relational databases. I also examine object-relational mapping in detail and give a comparison between the two types of databases.

Chapter 4 introduces some advanced topics in object modeling. They include abstract classes, multiple inheritance, interfaces, inner classes, collections, packages, and components. These are extensions to the basic object concepts. I also discuss the reverse engineering of object designs and the identification of irreducible patterns, which is presented in Chapter 2.

Chapter 5 describes modeling the dynamic behavior of objects. I discuss use case analyses and object sequence diagrams. I also introduce the important concepts of client/server and distributed objects. For distributed objects, I cover interface definition, and the Common Object Request Broker Architecture (CORBA) standard and its operational mechanisms.

Then in Chapter 6 I present various interface design patterns. These patterns are intimately related to the dynamic behaviors of their constituent objects. Such behaviors are documented with sequence diagrams. I also discuss interface patterns related to CORBA objects.

In Chapter 7 I elevate the discussion to the system level. I describe various object-oriented architectures, including procedural processing systems, client/server systems, layered systems, multi-tier systems, agents, and aggregations and federations. Note that the distinction between architecture and design is not absolute. In architecture we are more concerned with the coordination between components, overall system performance, and scaling properties. In design we focus on the details within a component, an interface, or a subsystem.

Chapter 8 gives summaries and notes for the preceding chapters, whereas Chapter 9 provides answers to all exercises.

The integrated case studies serve as real-life examples to illustrate the practical applications of the concepts. They appear at the ends of various chapters, culminating in Chapter 7 with a discussion of their system architectures. Readers are highly recommended to work through them in some detail. A concrete understanding of the basic concepts can only be built through hands-on design and implementation.

Sections with an asterisk after their titles may be skipped during the first reading. They are topics with somewhat narrower interests. Readers who are primarily interested in software system architecture may proceed directly to Chapter 7, which can be read as a survey of different architectural patterns.

Finally, the appendices provide various reference information. In particular, Appendix A summarizes UML notations, followed by a quick look-up table to all object designs appearing in the main text and exercises. Appendix B provides a list of code samples for each chapter. Appendix C lists the features of various object-oriented languages.

Online Resources

Fully functional code samples are available from http://www.awl.com/cseng/. The code samples have more than 40,000 source lines and are all written in Java. They cover nearly all examples described in the main text and most case studies. Appendix B lists the sample code for the chapters. Studying the code will help solidify the reader's understanding of the designs. Readers are encouraged to extend and enrich the sample code. Furthermore, students in courses on object-oriented programming languages may implement the designs appearing in the chapter exercises as additional exercises.

Acknowledgments

I thank my wife Monica for her patience and encouragement during this book project. I wish to thank Perry Cole, Dave Collins, Stevan Mrdalj, and Atma Sutjianto for their insightful comments and suggestions on the manuscript. Special thanks are due to Ross Venables, Paul Becker, and their colleagues at the editorial office for their efficient handling of the manuscript. The feedback from Amy Yuan and other students at the University of Maryland on an early version of the manuscript is gratefully acknowledged.

Contents

CHAPTER 1

Basic Concepts

Traditional procedural programming implies a flow of steps. From step 1, step 2, to step N, the whole process can be represented by a flow chart, which may include decision points and branches. However, as computer programs are applied to more complex situations, such as interactive user interfaces and systems with interconnecting components, the keeping of all possible procedural branches in a number of programs becomes a formidable task. The situation is especially serious when a software development project involves a sizable team of people, who have to coordinate their subtasks such that the integrated system works seamlessly.

To overcome these difficulties, breaking a computer program into relatively self-contained pieces is critical. Often, such a decomposition reveals the intrinsic logical structure of the underlying problem. This leads to *object-oriented programming*.

This chapter discusses the basic concepts in object-oriented programming, which include object, class, association, aggregation, servant class, and inheritance.

1.1 THE NATURE OF OBJECTS

Objects are abstractions of physical entities or conceptual things. An object has states and an inherent identity. It attains certain behavior through a set of predefined operations, which may access or change its state.

An object encapsulates its properties (called *attributes*) and the operations that access or change those properties. The state of the object is determined by the values of its attributes. The object state is set by the object's operations. The inclusion of operations distinguishes objects from mere data structures. Operations are identified by their names and signatures (input, output, return arguments and their types). The implementations of operations are called *methods* in Java (or *member functions* in C++).

An object class is the abstract descriptor of a set of object instances. It describes a set of object instances that share the same attributes, operations, and relationship

with other objects. For example, the Person class in Figure 1-1 contains attributes name and age, as well as operations for changing the name and incrementing the age. The Person class does not describe any specific person. The object instances of the Person class, on the other hand, have the information for actual persons.

For a physical entity, such as a person, the corresponding attributes are easily identified. Its operations, however, depend on how the object should behave in its environment and how it interacts with other objects. Determining the behavior of objects is a key to good object-oriented programming.

Objects corresponding to conceptual things also encapsulate attributes and operations. The only difference is that the attributes are properties or states that belong to a process, transaction, event, and so forth, rather than a physical object. A trade object example is shown in Figure 1-2.

Through abstraction, an object class generalizes a few specific object instances to a host of similar instances, thereby allowing efficient use of the implementation of its operations.

Although it is not our goal to map object designs to specific programming languages such as C++ or Java, it is beneficial to know how an object class is represented by such a language. Figure 1-3 shows the pseudocode for the Trade class. (The code is presented in Java style, but does not carry visibility modifiers like `public` or `private`. In other words, the default visibility is used.) The operation `calculatePrice()` retrieves the unit price (probably from a database), then calculates the total price, which is a property of the trade itself.

The term *object* is often used to indicate either an object class or an object instance. One simply needs to refer to the context to find out which meaning an "object" takes. Moreover, in performing object design, one often has to keep both meanings in mind to understand fully the relationships between objects.

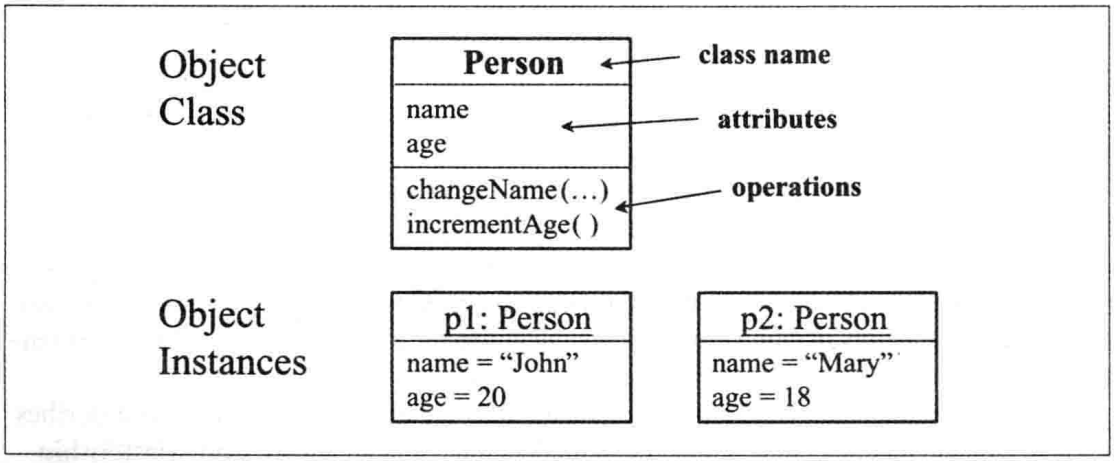

FIGURE 1-1. *Object classes and instances from physical objects.*

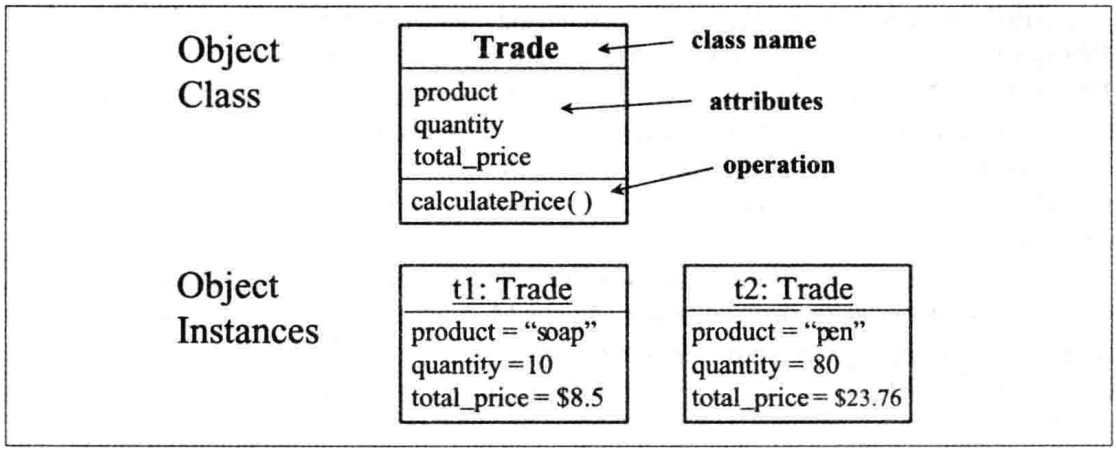

FIGURE 1-2. *Conceptual objects.*

```
class Trade {
 String product;
  int quantity;
 double total_price;

  // operations
   void calculatePrice( ) {
   // Retrievesunit_price, then
   // total_price = unit_price * quantity
   // Discount may apply for large quantities.
  }
}
```

FIGURE 1-3. *Java-like pseudocode for the Trade class.*

1.2 UNIFIED MODELING LANGUAGE

When we discuss objects, it is natural to present them using certain structured notations, as we have done in Figures 1-1 and 1-2. This has proved to be an effective way of modeling objects and documenting object designs.

The Unified Modeling Language (UML) is a general-purpose modeling language designed to specify, visualize, and document the artifacts of a software system. The visual notations of UML are particularly suited for object-oriented software designs.

UML is a unification of previous modeling methods such as Booch [Booch 1994], OOSE (object-oriented software engineering) of Jacobson [Jacobson et al. 1992], and OMT (object modeling technique) of Rumbaugh [Rumbaugh 1991]. The development of UML began in October 1994, when Grady Booch and Jim Rumbaugh of Rational Software Corporation began their work on unifying the Booch and OMT methods. In the fall of 1995, Ivar Jacobson joined the unification effort and merged in the OOSE method.

In 1997, the UML Partners consortium was formed. The consortium produced UML 1.0, which was submitted to the Object Management Group (OMG). Further inputs from several other companies were later incorporated to produce UML 1.1 in September 1997. UML 1.1 has been submitted to the OMG for adoption [UML 1997].

Compared with previous modeling methods, UML has the following two advantages:

1. A unified standard of semantics and notations brings regularity and stability to the software industry. Companies can make greater reuse of previous object designs based on a mature modeling language. Tool developers can focus on enhancing features rather than catching up with several evolving standards.
2. The joint effort yielded improvements from previous methods. It also helped to strike a balance between expressiveness and simplicity. It allows the object modeling method to evolve as a single body rather than as separate efforts, eliminating the potential for confusing differences.

Today, many companies are incorporating UML as a standard in their development processes and products. UML can be used to model both object structures and behavior. Structural models (or static models) focus on object classes, attributes, and the relationship between objects. Behavioral models (or dynamic models) emphasize object interaction, collaboration, and states.

Next we briefly describe the basic features of UML, which are used to illustrate our discussion. Other more involved topics are covered in future sections when needed. We note that UML contains a rich set of semantics and notations, and we refer you to other reference books [Fowler et al. 1997, Page-Jones 2000].

1.3 UML NOTATION BASICS

We have already seen the notation for object classes in Figures 1-1 and 1-2. Here we specify it more precisely. As shown in Figure 1-4, a class (long form) is represented by a solid-outline rectangle with three horizontal compartments. The name of the class