UML与面向对象设计影印丛书

对象 组件 框架 与UML应用

OBJECTS, COMPONENTS, AND FRAMEWORKS WITH UML

THE CATALYSIS℠ APPROACH

DESMOND FRANCIS D'SOUZA
ALAN CAMERON WILLS 编著

OBJECT TECHNOLOGY SERIES
BOOCH
JACOBSON
RUMBAUGH
ADDISON-WESLEY
SERIES EDITORS

科学出版社
www.sciencep.com

UML与面向对象设计影印丛书

对象 组件 框架与UML应用

Desmond Francis D'souza
Alan Cameron Wills　编著

科 学 出 版 社
北 京

内 容 简 介

本书介绍的是如何利用对象、框架和 UML 来设计和构建基于组件的软件系统并实现对系统的重用。全书共由 16 章组成，分为概述、对象建模、对象分析及设计、实施 Catalysis 应用等五个部分。不但内容详尽，而且循序渐进，非常有利于学习。

本书适合系统分析、设计人员阅读。

图书在版编目(CIP)数据

对象 组件 框架与 UML 应用=Objects Components, and Frameworks with UML/（美）苏泽（Souza,D.F.）（美）威尔斯（Wills,A.C.）著.—影印本. —北京：科学出版社，2003

ISBN 7-03-011408-6

Ⅰ.对… Ⅱ.①苏…②D…③威…④A… Ⅲ.面向对象语言，UML—程序设计—英文 Ⅳ.TP312

中国版本图书馆 CIP 数据核字（2003）第 030811 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：50 3/4
印数：1—2 000 字数：972 000

定价：80.00 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

Businesses, and the worlds in which they operate, are always changing. Nearly all businesses use software to support the work they do, and many of them have software embedded in the products they make. Our software systems must meet those business needs, work properly, be effectively developed by teams, and be flexible to change.

First Requirement of Software: Integrity

The first two points are old news; since 1968, the software community has been inventing methods to help understand and meet the business requirements. The average piece of desktop software may work properly in some average sense; perhaps the occasional bug is better than waiting until developers get it perfect. On the other hand, these days so much more software is embedded—in everything from vehicle brakes, aircraft, and smart cards to toasters and dental fillings.[1] Sometimes we should care deeply about the integrity of that software. The techniques described in this book will help you get the requirements right and implement them properly.

Second Requirement of Software: Team Development

To enable development by distributed teams, work units must be separated and partitioned with clear dependencies, architectural conventions and rules must be explicit, and interfaces must be specified unambiguously. Components will get assembled by persons different from the developers, potentially long after they are built; the relationships between the implementations, interface specifications, and eventual user requirements must be testable in a systematic way.

1. Teeth have historically been a central part of a user's interface.

This book's techniques will help you build work packages and components with these properties.

Third Requirement of Software: Flexibility

To stay competitive, businesses must continually provide new products and services; thus, business operations must change in concert. Banks, for example, often introduce new deals to lure customers away from the competition; today, they offer more new services via the phone and the Internet than through branch offices. Flexible software, which changes with the business, is essential to competitiveness.

Flexibility means the ability not only to change quickly but also to provide several variants at the same time. A bank may deploy the same basic business system globally but needs to be able to adapt it to many localized rules and practices. A software product vendor cannot impose the same solution on every customer nor develop a solution from scratch each time. Instead, software developers prefer to have a configurable family of products.

This book's techniques will help you partition and decouple software parts in a systematic way.

Flexible Software

The key to making a large variety of software products in a short time is to make one piece of development effort serve for many products. Reuse does not mean that you can cut-and-paste code: The proliferation that results, with countless local edits, rapidly becomes an expensive maintenance nightmare.

The more effective strategy is to make generic designs that are built to be used in a variety of software products. Such reusable assets include code as well as models, design patterns, specifications, and even project plans.

The following are two key rules for building a repertoire of reusable parts.

- They should not be modified by the designers who use them. You want only one version of each part to maintain; it must be adaptable enough to meet many needs, perhaps with customization but without modification.
- They should form a coherent kit. Building things with a favorite construction toy, such as Legos, is much easier and faster than gluing together disparate junk you found in the back of the garage. The latter may be parts, but they weren't designed to fit together.

Reusable parts that can be adapted, but not modified, are called *components;* they range from compiled code without program source to parts of models and designs.

Families of Products from Kits of Components

Hardware designers have been building with standardized components for years. You don't design one new automobile; rather, you design a family of them. Variations are made by combining a basic set of components into different configurations. Only a few components are made specifically for one product. Some are made for the family of products, others are shared with previous families, and still others are made by third parties and shared with other makes of cars.

We can do the same thing with software, but we need technologies for building them, and assembling them, into products as well as methods for designing them.

Component Technology

For a component to be generic, you must provide ways that your clients' designers can specialize it to their needs. The techniques include parameters passed when a function is called; tables read by the component; configuration or deployment options on the component; *plug-points*—a place where the component can be plugged into a variety of other components; and frameworks, such as a workflow system, into which a variety of components can be plugged.

Object-oriented (OO) programming underscores the importance of pluggability, or *polymorphism*: the art and technology of making one piece of software that can be coupled with many others. People have always divided programs into modules; but the original reasons were meant to divide work across a team and to reduce recompilation. With pluggable software, the idea is that you can combine components in different ways to make different software products—in the same way that hardware designers can make many products from a kit of chips and boards—and can do so with a range of delayed binding times (see Figure P.1).

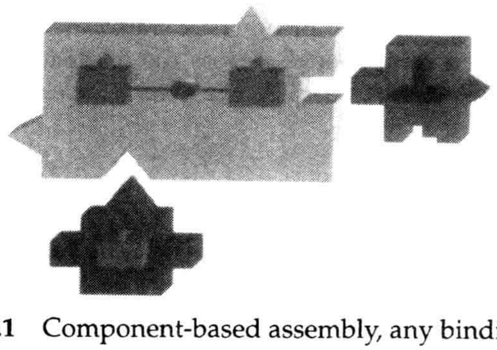

Figure P.1 Component-based assembly, any binding time.

The other great idea reemphasized in OO programming is the separation of concerns. The idea is that each object or component, or reusable part, should have one responsibility and that its design should be as decoupled (independent) as possible from designs and even the existence of other components.

Both these ideas work whether or not you use an OO programming language and whether you are talking about objects in programming, large distributed systems, or departments in a business.

Where Do We See Components?

On a small scale, pluggable user-interface widgets form components. Several such kits come with visual builders, such as Visual Basic, which help you plug the components together. Kits also may extend to small parts outside the user interface domain (for example, VisualAge and JavaBeans). These components all work within one executable program.

On a larger scale, self-contained application programs can be driven by each other; object linking and embedding/Component Object Model (OLE/COM), UNIX pipes and signals, and Apple events allow this to happen. Communicating components can be written in different languages, and each can execute in its own space.

On a still larger scale, components can be distributed between different machines. Distributed COM (DCOM) and Common Object Request Broker Architecture (CORBA) are the latest technologies; various layers underneath them, such as TCP/IP, provide for more primitive connections. When you deploy components on this scale, you must worry about new kinds of distributed failures and about economies of object location. Workflow, replication, and client-server, or n-tier architectures, provide frameworks into which this scale of component can fit. Again, there are tools and specialized languages that can be used to build such systems. Enterprise JavaBeans and COM+ are newer technologies that relieve the component developer of many of the worries of working with large-grained server-side shared components.

A component, on the larger scale, often supports a particular business role played by an individual or department with responsibility for a particular function. Businesses talk increasingly of open federated architectures, in which the structure of distributed components mirrors the organizational structure of the business. When reorganizing a business, we need to be able to rewire software in the same way.

Challenges of Component-Based Development

The technology of component-based systems is becoming fairly well established; not so the methods to develop them. To be successful, serious enterprise-level development needs clear, repeatable procedures and techniques for development, well-defined and standard architectures, and unambiguous notations whereby colleagues can communicate about their designs.

A key technique for building a kit of components is that you must define the interfaces between the components very clearly. This brings us back to integrity. If we are to plug together parts from different designers who don't know one another, we must be very clear about what the contract across the connection is: what each party should provide to and expect of the other.

In component technologies, such as COM, CORBA, and JavaBeans, the emphasis is on defining interfaces. (The idea has a long history, however, stretching back to experimental languages such as CLU in the 1970s.) The same thing is true no matter what the technology: UNIX pipes, workflow, RPC, common access to a database, and the like. Whenever a part can fit into many others, you must define how the connection works and what is expected of the components that can be plugged in.

In Java or CORBA, though, *interface* means a list of function calls. This definition is inadequate for good design on two counts. First, to couple enterprise-scale components, we need to talk in bigger terms: A connection might be a file transfer or a database transaction involving a complex dialog. So we need a design notation that doesn't always have to get down to the individual function calls; and it should be able to talk about the messages that come out of a component as well as those that go in. JavaBeans (and Enterprise JavaBeans) go some of the way in this direction. In Catalysis, we talk about *connectors* to distinguish higher-level interfaces from basic function calls.

Second, function calls that are described only by their parameter signatures do not tell enough about the expected behavior. Programming languages do not provide this facility because they are not intended to represent designs; but we need to write precise interface descriptions. The need for precision is especially acute because each component may interface with unknown others. In the days of modular programming, designers of coupled modules could resolve questions around the coffee machine; in a component-based design, the components may have been put together by two people and assembled independently by a third.

To develop a coherent kit of components, we must begin by defining a common set of connectors and common models of what the components talk to one another about. In a bank, for example, there is no hope of making the components reconfigurable unless all of them use the same definition of basic concepts such as customer, account, and money (at their connectors, if not internally).

Once a common set of interfaces and a common architectural framework are laid down, many designers can contribute components to the kit. Products can then be assembled from the components (see Figure P.2).

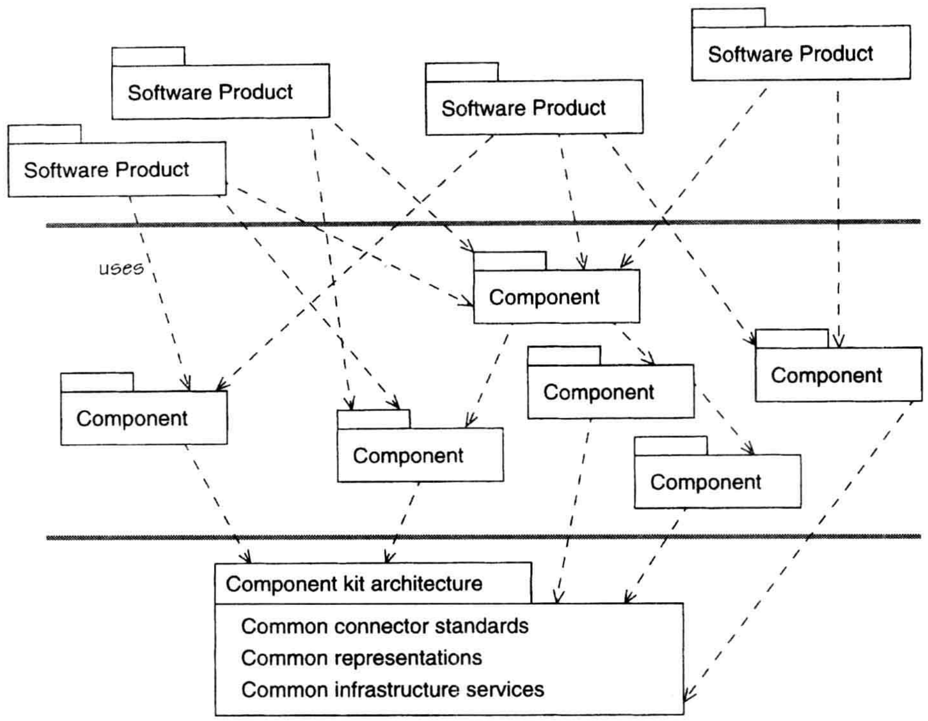

Figure P.2 Products can be assembled from components supplied by many sources.

What Does Catalysis Provide?

If this component-based scenario seems far-fetched, recall the fate of Babbage's Analytical Engine. He couldn't make it work because it had so many parts and they didn't have the machining techniques to make the parts fit together well enough. Today's machining has enabled working versions to be made. As our software industry improves its skills and consistency in making matching parts, we will also make products from components.

This book gathers together some of the techniques we see as necessary for that movement into a coherent kit. To make component-based development work, we need our best skills as software designers, and we need to reorganize the ways in which software is produced.

The techniques and method in Catalysis provide the following:

- *For component-based development*: How to precisely define interfaces independent of implementation, how to construct the component kit architecture and the component connectors, and how to ensure that a component conforms to the connectors.

- *For high-integrity design*: Precise abstract specifications and unambiguous traceability from business goals to program code.
- *For object-oriented design*: Clear, use case driven techniques for transforming from a business model to OO code, with an interface-centric approach and high quality assurance.
- *For reengineering*: Techniques for understanding existing software and designing new software from it.

Catalysis and Standards

Catalysis uses notation based on the industry standard Unified Modeling Language (UML) now standardized by the Object Modeling Group (OMG). Both authors have been involved in the OMG standards submissions for object modeling; Desmond's company helped define and cosubmit UML 1.0 and 1.1.

Catalysis has been central to the component-specification standards defined by Texas Instruments and Microsoft, the CBD-96 standards from TI/Sterling, and services and products from Platinum Technology; it has been adopted by several companies as their standard approach for UML-based development. It fits the needs of Java, JavaBeans, COM+, and CORBA development and supports the approach of RM-ODP. It also supports systematic development based on use cases.

Where Does Catalysis Come From?

Catalysis is based on, and has helped shape, standards in the object modeling world. It is the result of the authors' work in development, consulting, and training and is based on experience with clients from finance, telecommunications, aerospace, GIS, government, and many other fields.

Many ideas in Catalysis are borrowed from elsewhere. The Bibliography section lists many of the specific references. We can identify and gratefully acknowledge general sources of the principal features of Catalysis.

- We began applying rigorous methods to object analysis with OMT [Rumbaugh 91]. Integrating snapshots, transactions, state models, treating system operations and analysis models separately from design classes, and the basic ideas of refinement of time granularity date from Desmond's work at this time.
- The rigorous aspects (specifications, refinement, and the influence of VDM and Z) were seen particularly in some previous OO development methods: Fusion [Coleman93], Syntropy [Cook 94], and Bon [Meyer88]. Our interest in applying rigorous methods, such as VDM and Z to objects goes back to Alan's Ph.D. thesis [Wills91].
- Collaborations as first-class design units were first introduced in Helm, Holland, and Gangopadhyay's "contracts" and developed in Trygve Reenskaug's [Reenskaug95] method and tool OORAM.

- Abstract joint actions come from Disco [Kurki-Suonio90], the OBJ tradition [Goguen90], and database transactions as well as from the general notion of the Objectory use case.
- Component connectors have been mentioned in a variety of patterns in recent years. They date back to Wong's *Plug and Play Programming* work [Wong90], previous work (mostly in the Smalltalk arena) on code frameworks, and architecture work on components and connectors [Shaw96b].
- Process patterns are a corruption of work by several of the contributors to the Pattern Languages of Programming conferences.

During the development of Catalysis, we have also had a great deal of input and feedback from many clients and fellow consultants, teachers, and researchers (see Thank You section of this Preface).

How to Read This Book

Don't read it all in one night. If you think this is a bit long for a Preface, wait until you see the rest of the book. What background will you need? Some basic knowledge of UML, OMT, Booch, or Fusion modeling will help; the succinct UML summary by Martin Fowler is quite readable [Fowler98]. If you already know UML, take an early look at the UML perspective in the Appendixes.

Begin with Chapter 1—a tour that leads you through the essence of a design job. Along the way it bumps into all the main Catalysis techniques and ends with a summary of our approach and its benefits. Then read the introduction to each subsequent part (I–V) to get a feel for the book's structure. Most of the subsequent chapters are designed so that you can read the first sections and the summary at the end and then skip to the next chapter. After you've gone through the book this way, go back and dig down into the interesting stuff.

There are places in the book where we discuss some of the darker corners of modeling, and it's safe to skip these sections. We have marked most of these sections with this icon. There are also places where we illustrate implementations using Java; if this is new to you, you can usually skip these bits as well.

Chapters 2, 3, and 4 are groundwork: They tell you how to make behavioral models and what they mean and don't mean. Chapter 5 is essential: how to document a design. Chapter 6, Abstraction, Refinement, and Testing, is about how to construct a precise relationship between a business model and the program code. Chapters 7 through 9 (Using Packages, Composing Models and Specifications, and Model Frameworks and Template Packages) deal with breaking models into reusable parts and composing them into specifications and designs. Chapters 10, 11, and 12 (Components and Connectors, Reuse and Pluggable Design: Frameworks in Code, and Architecture) are about building enterprise-scale software from reusable components. Chapters 13 through 16 are about the process of applying Catalysis, exploring a case study in considerable detail. Depending on your role, here are some suggested routes.

• *Analysts*: Mainstream OO analysis is difficult if you are used to structured methods. In some ways our approach is simpler: You explore system-level scenarios, describe the system operations, capture terms you use in a static model of the system, and then formalize operations using this model. In other ways, our approach is more difficult; we do not like fuzzy and ambiguous analysis documents, so some of the precision we recommend may be a bit unfamiliar for early requirements' activities. Read Chapters 1 through 7, 9, and 13 through 15.

• *Designers*: Object-oriented design is as novel as OO analysis. Again, in some ways our approach is simpler. You start with a much clearer description of the required behaviors, and there is a default path to basic OO design that you can follow (see Pattern 16.8, Basic Design). For doing component-based design, you will use the techniques of an analyst, except at the level of your design components.

If you are already an OO designer, be prepared for a different focus. First, you understand the behavior of a large-grained object (system, component) as a single entity. Then you build an implementation-independent model of its state, and then design its internal parts and the way they interact. You strictly distinguish type/interface from class and always write an implementation class against other interfaces. Read Chapters 1 through 6 (omit sections that go into specification details), 7, 9, 10 through 12, and 16.

• *Implementors*: OO implementation should become easier when the task of satisfying functional requirements has been moved into the design phase. Implementation decisions can then concentrate on exploiting the features of a chosen configuration and language needed to realize all the remaining requirements.

• *Testers*: Testing is about trying to show that an implementation does not meet its specification by running test data and observing responses. Specifications describe things that range from what a function call should do to which user tasks the system must support; the way to derive tests varies accordingly. Read Chapters 1 through 6. Also, read about QA (see Section 13.1, Model, Design, Implement, and Test—Recursively; and Section 13.2, General Notes on the Process), and insist that it be followed well before testing.

• *Project managers*: Consider your goals for using components or objects carefully and the justifications for building flexible and pluggable parts (Chapter 10). Watch out for the project risks, often centered on requirements and infrastructure (Chapter 13). Together with the architect, design and follow the evolution of the package structure (Chapter 7) and how it gets populated; if there is such a thing as development architecture, that is it. Recognize the importance of a precise vocabulary shared by the team (Chapters 2 and 3). Read Chapters 1 through 5, and (optionally) Chapters 6, 7, 12, and 13. Consider starting with "Catalysis lite" (www.catalysis.org).

• *Tool builders*: Catalysis opens new opportunities for automated tool support in modeling, consistency checking, traceability, pattern-based reuse, and project management. Read the book.

• *Methods and process specialists*: Some of what we say is new; the parts fit together, and the core is small, so look closely. Read the book.

• *Students and teachers*: There is material in this book for several semester-long courses and several research projects, and perhaps even for course-specific books. Few courses are based on a rigorous model-based approach to software engineering. We have successfully used the material in this book in several one-week courses and workshops and know of several universities that are adopting it. If you want to use some of the illustrations in this book in your presentations, you need to have permission. Please contact Addison Wesley Longman, Inc. at the address listed on the copyright page.

• *Others*: The activities and techniques in this book apply to both large and small projects, with different emphases and explicit deliverables, and to business modeling, bidding on software projects, out-tasking, and straightforward software development, even though the rigor in our current description might intimidate some. See www.catlysis.org.

Where to Find More

When you've finished the book and are eager for more, there is a Catalysis Web site—www. catalysis.org—that will provide additional information and shared resources, potentially including the following:

- Example models, specification, documentation, and frameworks
- Discussion of problems this book has not yet fully addressed: concurrency, distribution, business process models, and so on
- Web-based discussion forums and mailing lists for users, teachers, consultants, researchers, tire-kickers, and lost souls to share experiences and resources
- Free as well as commercial tools that support the Catalysis development and modeling techniques
- On-line versions of the book and development process patterns
- Modeling exercises and solutions for university use
- Resources to help others use and promote Catalysis, including short presentations to educate fellow modelers, designers, and managers; summary white papers that can be handed out on Catalysis; and so on.

In addition, there are Web sites for each author's company. Each contains a great deal of interesting material, which will continue to be updated:

- http://www.iconcomp.com/catalysis—ICON Computing, a Platinum Technology company (www.platinum.com)
- http://www.trireme.com/catalysis—TriReme International Limited

Thank You

Thanks to our editors—Mike Hendrickson and Debbie Lafferty—for their patience and encouragement; and to our production coordinator, Marilyn Rash, and her team—Betsy Hardinger, copyeditor; Maine Proofreading Services; and Publisher's Design and Production Services for expert, speedy art rendering and typesetting.

Our book reviewers bravely hacked through initial drafts and greatly helped improve this book. To Joseph Kiniry (a heroic last-ditch effort), Doug Lea, Jennie Beckley, Ted Velkoff, Jay Dunning, and Gerard Meszaros—many thanks.

Several others provided comments and ideas: John McGehee, Stuart Kent, Mike Mills, Richard Mitchell, Keith Short, Bill Gibson, Richard Veryard, Ian Maung, Dale Campbell, Carol Kasmiski, Markus Roësch, Larry Wall, Petter Graff, and John Dodd. Aamod Sane and Kevin Shank helped sort out issues with nested packages. We would also like to thank, for useful technical discussions and support: Balbir Barn, Grady Booch, John Cameron, John Cheesman, Steve Cook, John Daniels, Chris Dollin, John Fitzgerald, Ian Graham, Brian Henderson-Sellars, Benedict Heal, John Hogg, Trevor Hopkins, Iain Houston, Cliff Jones, Kevin Lano, Doug Lea, Clive Mabey, Tobias Nipkow, David Redmond-Pyle, Howard Ricketts, John Robinson, Jim Rumbaugh, Susan Stepney, Charles Weir, Anthony Willoughby, and Jim Woods.

We are very grateful to many others for their feedback and suggestions. For their encouragement and support, thanks to Clive Menhinick at TriReme and the team at ICON; and, from Desmond, a very special thanks to Mama, Tootsie and Clifford, and to Tina's parents. Alan would like to thank his remaining friends.

Should these good folks deny any responsibility for the final product, we will gladly take the blame for all inconsistencies and omissions; we know there are some lurking in these pages, and hope you find this work useful despite them.

Desmond Francis D'Souza *Alan Cameron Wills*

Objects, Components, and Frameworks with UML

The Catalysis ^{SM} ***Approach***