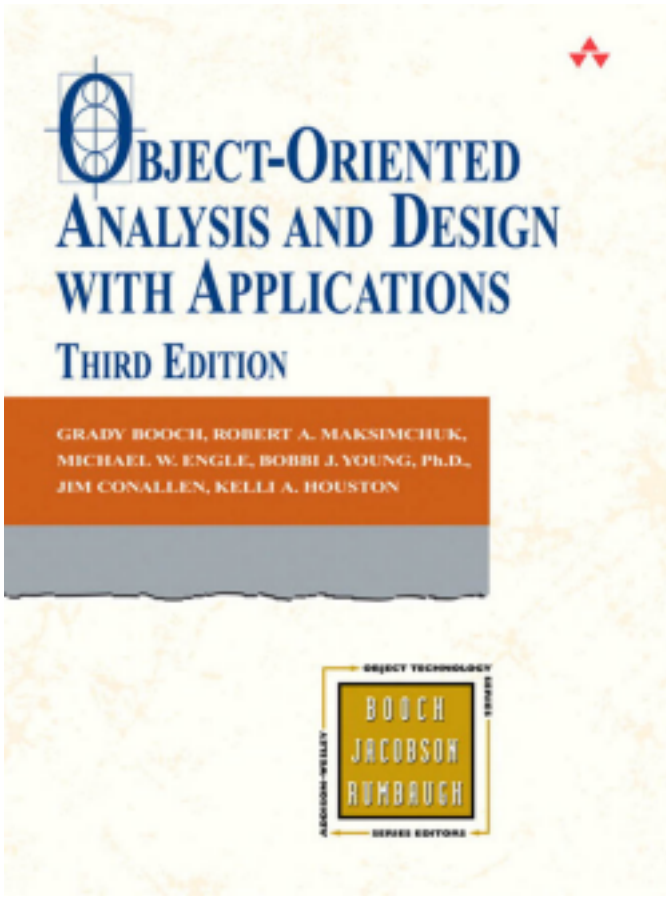


Object-Oriented Analysis and Design with Applications

Third Edition

The Addison-Wesley Object Technology Series

Grady Booch, Ivar Jacobson, and James Rumbaugh, Series Editors For more information, check out the series web site at www.awprofessional.com/otseries.

Ahmed/Umrysh, *Developing Enterprise Java Applications with J2EE™ and UML*

Arlow/Neustadt, *Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML*

Arlow/Neustadt, *UML 2 and the Unified Process, Second Edition* Armour/Miller,*Advanced Use Case Modeling: Software Systems* Bellin/Simone, *The CRC Card Book*

Bergström/Råberg, *Adopting the Rational Unified Process: Success with the RUP*

Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools* Bittner/Spence, *Use Case Modeling*

Booch, *Object Solutions: Managing the Object-Oriented Project* Booch, *Object-Oriented Analysis and Design with Applications, 2E* Booch/Bryan, *Software Engineering with ADA, 3E*

Booch/Rumbaugh/Jacobson, *The Unified Modeling Language User Guide, Second Edition*

Box et al., *Effective COM: 50 Ways to Improve Your COM and MTS based Applications*

Buckley/Pulsipher, *The Art of ClearCase® Deployment* Carlson, *Modeling XML Applications with UML: Practical e-Business Applications*

Clarke/Baniassad, *Aspect-Oriented Analysis and*

Design Collins, *Designing Object-Oriented User Interfaces*

Conallen,*Building Web Applications with UML, 2E*

Denney, *Succeeding with Use Cases*

D'Souza/Wills, *Objects, Components, and Frameworks with UML: The Catalysis(SM) Approach*

Douglass, *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*

Douglass,*Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*

Douglass,*Real Time UML, 3E: Advances in The UML for Real-Time Systems*

Eeles et al., *Building J2EE™Applications with the Rational Unified Process* Fowler, *Analysis Patterns: Reusable Object Models*

Fowler, *UML Distilled, 3E: A Brief Guide to the Standard Object Modeling Language*

Fowler et al., *Refactoring: Improving the Design of Existing Code* Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*

Gomaa, *Designing Software Product Lines with UML*

Heinckiens,*Building Scalable Database Applications: Object-Oriented Design, Architectures, and Implementations*

Hofmeister/Nord/Dilip,*Applied Software Architecture*

Jacobson/Booch/Rumbaugh, *The Unified Software Development Process* Jacobson/Ng,*Aspect-Oriented Software Development with Use Cases* Jordan,*C++ Object Databases: Programming with the ODMG Standard*

Kleppe/Warmer/Bast, *MDA Explained: The Model Driven Architecture™: Practice and Promise*

Kroll/Kruchten, *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*

Kruchten, *The Rational Unified Process, 3E: An Introduction* LaLonde, *Discovering Smalltalk*

Lau, *The Art of Objects: Object-Oriented Design and Architecture* Leffingwell/Widrig, *Managing Software Requirements, 2E: A Use Case Approach*

Manassis,*Practical Software Engineering: Analysis and Design for the .NET Platform*

Marshall,*Enterprise Modeling with UML: Designing Successful Software through Business Analysis*

McGregor/Sykes, *A Practical Guide to Testing Object-Oriented Software* Mellor/Balcer, *Executable UML: A Foundation for Model-Driven Architecture*

Mellor et al., *MDA Distilled: Principles of Model-Driven Architecture* Naiburg/Maksimchuk, *UML for Database Design*

Oestereich, *Developing Software with UML, 2E: Object-Oriented Analysis and Design in Practice*

Page-Jones, *Fundamentals of Object-Oriented Design in UML* Pohl, *Object-Oriented Programming Using C++, 2E*

Pollice et al. *Software Development for Small Teams: A RUP-Centric Approach*

Quatrani,*Visual Modeling with Rational Rose 2002 and UML* Rector/Sells,*ATL Internals*

Reed, *Developing Applications with Visual Basic and UML* Rosenberg/Scott, *Applying Use Case Driven Object Modeling with UML: An Annotated e-Commerce Example*

Rosenberg/Scott, *Use Case Driven Object Modeling with UML: A Practical Approach*

Royce, *Software Project Management: A Unified Framework* Rumbaugh/Jacobson/Booch, *The Unified Modeling Language Reference Manual*

Schneider/Winters, *Applying Use Cases, 2E: A Practical Guide* Smith,*IBM Smalltalk*

Smith/Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*

Tkach/Fang/So, *Visual Modeling Technique*

Tkach/Puttick, *Object Technology in Application Development, Second Edition*

Unhelkar, *Process Quality Assurance for UML-Based Projects* Warmer/Kleppe, *The Object Constraint Language, 2E: Getting Your Models Ready for MDA*

White, *Software Configuration Management Strategies and Rational ClearCase®: A Practical Introduction*

The Component Software Series Clemens Szyperski, Series Editor

For more information, check out the series web site at www.awprofessional.com/csseries.

Cheesman/Daniels, *UML Components: A Simple Process for Specifying Component-Based Software*

Szyperski,*Component Software, 2E: Beyond Object-Oriented Programming*

Object-Oriented Analysis and Design with Applications

Third Edition

Grady Booch
Robert A. Maksimchuk
Michael W. Engle
Bobbi J. Young, Ph.D.
Jim Conallen
Kelli A. Houston

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris •
Madrid Capetown • Sydney • Tokyo • Singapore • Mexico City

Visit us on the Web: www.awprofessional.com

Library of Congress Cataloging-in-Publication Data

Object-oriented analysis and design with applications / Grady Booch...[et al.]. — 3rd ed.
p. cm.
Rev. ed. of: Object-oriented analysis and design with applications / Grady Booch, 2nd ed.
Includes bibliographical references and index.
ISBN 0-201-89551-X (hardback : alk. paper)
1. Object-oriented programming (Computer science) I. Booch, Grady. II. Booch, Grady. Object-oriented analysis and design with applications.

QA76.64.B66 2007
005.1'17—dc22 2007002589

ISBN 0-201-89551-X
Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
First printing, April 2007

To Jan
my friend, my lover, my
wife —Grady

This page intentionally left blank

Contents

Sidebars

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 6

Chapter 8

Chapter 9

Chapter 12

Preface

> Mankind, under the grace of God, hungers for spiritual peace, esthetic achievements, family security, justice, and liberty, none directly satisfied by industrial productivity. But productivity allows the sharing of the plentiful rather than fighting over scarcity; it provides time for spiritual, esthetic, and family matters. It allows society to delegate special skills to institutions of religion, justice, and the preservation of liberty.
>
> HARLAN MILLS
> *DPMA and Human Productivity*

As computer professionals, we strive to build systems that work and are useful; as software engineers, we are faced with the task of creating com plex systems in the presence of constrained computing and human resources. Object-oriented (OO) technology has evolved as a means of managing the complexity inherent in many different kinds of systems. The object model has proven to be a very powerful and unifying concept.

Changes to the Second Edition

Since the publication of the second edition of *Object-Oriented Analysis and Design with Applications*, we have seen major technological advances. This list includes some highlights, among many others.

- High-bandwidth, wireless connectivity to the Internet is widely available.
- Nanotechnology has emerged and has started to provide valuable products.

- Our robots are cruising the surface of Mars.
- Computer-generated special effects have enabled the film industry to recre ate any world imaginable with complete realism.
- Personal hovercraft are available.
- Mobile phones have become pervasive to the point of being disposable.
- We have mapped the human genome.

- Object-oriented technology has become well established in the mainstream of industrial-strength software development.

We have encountered the use of the object-oriented paradigm throughout the world. However, we still encounter many people who have not yet adopted the object paradigm of development. For both of these groups, this revision of this book holds much value.

For the person new to object-oriented analysis and design (OOAD), this book gives the following information:

- The conceptual underpinnings of and evolutionary perspective on object orientation
- Examples of how OOAD can be applied across the system development lifecycle
- An introduction to the standard notation used in system and software devel opment, the Unified Modeling Language (UML 2.0)

For the experienced OOAD practitioner, the content herein provides value from a different perspective.

- UML 2.0 is still new to even experienced practitioners. Here you will see the key changes in the notation.
- More focus on modeling is provided, per feedback received about the previ ous edition.
- You can gain a great appreciation for "why things are the way they are" in the object-oriented world, from the Concepts section of the book. Many people may never have been exposed to this information on the evolution of the OO concepts themselves. Even if you have been, you may not have grasped its significance when you were first learning the OO paradigm.

There are four major differences between this edition and the previous publication.

1. UML 2.0 has been officially approved. Chapter 5, Notation, will introduce UML 2.0. To enhance the reader's understanding of this notation, we explic itly distinguish between its fundamental and advanced elements.

2. This edition introduces some new domains and contexts in the Applications chapters. For example, the application domains range broadly across vari ous levels of abstraction from high-level systems architecture to the design details of a Web-based system.
3. When the previous edition was published, C++ was relatively new, as was the very concept of OO programming. Readers tell us that this emphasis is no longer a primary concern. There is an abundance of OO programming and technique books and training available, not to mention additional pro gramming languages designed for OO development. Therefore, most of the coding discussions have been removed.
4. Finally, in response to requests received from readers, this edition focuses

much more on the modeling aspects of OOAD. The Applications section will show you how to use the UML, with each chapter emphasizing one phase of the overall development lifecycle.

Goals

This book provides practical guidance on the analysis and design of object oriented systems. Its specific goals are the following:

- To provide a sound understanding of the fundamental concepts and historical evolution of the object model
- To facilitate a mastery of the notation and process of object-oriented analysis and design
- To teach the realistic application of object-oriented analysis and design within a variety of problem domains

The concepts presented all stand on a solid theoretical foundation, but this is primarily a pragmatic book that addresses the practical needs and concerns of software engineering practitioners, from the architect to the software developer.

Audience

This book is written for the computer professional as well as for the student.

- For the practicing systems and software developer, we show you how to effectively use object-oriented technology to solve real problems. ■ In your role as an analyst or architect, we offer you a path from requirements to implementation, using object-oriented analysis and design. We

develop your ability to distinguish "good" object-oriented architectures from "bad" ones and to trade off alternate designs when the perversity of the real world intrudes. Perhaps most important, we offer you fresh approaches to reasoning about complex systems.

- For the program manager, we provide insight on topics such as allocation of resources of a team of developers, software quality, metrics, and management of the risks associated with complex software systems.
- For the student, we provide the instruction necessary for you to begin acquiring several important skills in the science and art of developing complex systems.

This book is also suitable for use in undergraduate and graduate courses as well as in professional seminars and individual study. Because it deals primarily with a method of software development, it is most appropriate for courses in

software engineering and as a supplement to courses involving specific object-oriented programming languages.

Structure

The book is divided into three major sections—Concepts, Method, and Applications—with considerable supplemental material woven throughout.

Concepts

Section I examines the inherent complexity of software and the ways in which complexity manifests itself. We present the object model as a means of helping us manage this complexity. In detail, we examine the fundamental elements of the object model such as: abstraction, encapsulation, modularity, and hierarchy. We address basic questions such as "What is a class?" and "What is an object?" Because the identification of meaningful classes and objects is the key task in object-oriented development, we spend considerable time studying the nature of classification. In particular, we examine approaches to classification in other disciplines, such as biology, linguistics, and psychology, and then apply these lessons to the problem of discovering classes and objects in software systems.

Method

Section II presents a method for the development of complex systems based on the object model. We first present a graphic notation (i.e., the UML) for object-

oriented analysis and design, followed by a generic process framework. We also examine the pragmatics of object-oriented development—in particular, its place in the software development lifecycle and its implications for project management.

Applications

Section III offers a collection of five nontrivial examples encompassing a diverse selection of problem domains: system architecture, control systems, cryptanalysis, data acquisition, and Web development. We have chosen these particular problem domains because they are representative of the kinds of complex problems faced by the practicing software engineer. It is easy to show how certain principles apply to simple problems, but because our focus is on building useful systems for the real world, we are more interested in showing how the object model scales up to complex applications. The development of software systems is rarely amenable to cookbook approaches; therefore, we emphasize the incremental development of applications, guided by a number of sound

principles and well-formed models.

Supplemental Material

A considerable amount of supplemental material is woven throughout the book. Most chapters have sidebars that provide information on related topics. We include an appendix on object-oriented programming languages that summarizes the features of a few common languages. We also provide a glossary of common terms and an extensive classified bibliography that lists references to source mate rial on the object model.

A Note about Tools

Readers always ask about the tools used to create the diagrams in the book. Pri marily, we have used two fine tools for the diagrams: IBM Rational Software Architect and Sparx Systems Enterprise Architect. Why not use just one? The reality of the marketplace is that no tool does everything. The longer you do OOAD, you will eventually find some atypical "corner case" that no tool sup ports. (In that case, you may have to resort to basic drawing tools to show what you want.) But don't let those rare instances stop you from using robust OOAD tools such as those we mentioned.

Using This Book

This book may be read from cover to cover or it may be used in less structured ways. If you are seeking a deep understanding of the underlying concepts of the object model or the motivation for the principles of object-oriented development, you should start with Chapter 1 and continue forward in order. If you are prima rily interested in learning the details of the notation and process of object-oriented analysis and design, start with Chapters 5 and 6; Chapter 7 is especially useful to managers of projects using this method. If you are most interested in the practical application of object-oriented technology to specific problems, select any or all of Chapters 8 through 12.

Acknowledgments

This book is dedicated to my wife, Jan, for her loving support.

Through both the first and second editions, a number of individuals have shaped

my ideas on object-oriented development. For their contributions, I especially thank Sam Adams, Mike Akroid, Glenn Andert, Sid Bailin, Kent Beck, Dave Bernstein, Daniel Bobrow, Dick Bolz, Dave Bulman, Kayvan Carun, Dave Collins, Damian Conway, Steve Cook, Jim Coplien, Brad Cox, Ward Cunningham, Tom DeMarco, Mike Devlin, Richard Gabriel, William Genemaras, Adele Goldberg, Ian Graham, Tony Hoare, Jon Hopkins, Michael Jackson, Ralph Johnson, James Kempf, Norm Kerth, Jordan Kreindler, Doug Lea, Phil Levy, Barbara Liskov, Cliff Longman, James MacFarlane, Masoud Milani, Harlan Mills, Robert Murray, Steve Neis, Gene Ouye, Dave Parnas, Bill Riddel, Mary Beth Rosson, Kenny Rubin, Jim Rumbaugh, Kurt Schmucker, Ed Seidewitz, Dan Shiffman, Dave Stevenson, Bjarne Stroustrup, Dave Thomas, Mike Vilot, Tony Wasserman, Peter Wegner, Iseult White, John Williams, Lloyd Williams, Niklaus Wirth, Mario Wolczko, and Ed Yourdon.

A good part of the pragmatics of this book derives from my involvement with complex software systems being developed around the world at companies such as Alcatel, Andersen Consulting, Apple, AT&T, Autotrol, Bell Northern Research, Boeing, Borland, Computer Sciences Corporation, Contel, Ericsson, Ferranti, General Electric, GTE, Holland Signaal, Hughes Aircraft Company, IBM, Lockheed, Martin Marietta, Motorola, NTT, Philips, Rockwell Interna tional, Shell Oil, Symantec, Taligent, and TRW. I have had the opportunity to interact with literally hundreds of professional software engineers and their man agers, and I thank them all for their help in making this book relevant to real world problems.

A special acknowledgment goes to Rational for its support of my work. Thanks also to Tony Hall, whose cartoons brighten what would otherwise be just another stuffy technical book. Finally, thanks to my three cats, Camy, Annie, and Shadow, who kept me company on many a late night of writing.

—Grady Booch

First I want to thank God, without whom none of this would be possible. I want to thank my family, who, once again, had to deal with those long hours of my absence while working on this project. Thanks to my parents, who gave me their strong work ethic. Thanks to Mary T. O'Brien, who started it all by offering me this opportunity, and thanks to Chris Guzikowski for helping drive this to comple tion. To my fellow writers, thank you for agreeing to join me on this journey and for all your hard work and contributions toward this project. Last, but absolutely not least, my heartfelt thanks to Grady for all his work those many years ago, creating one of the original, foundational books on object-oriented analysis and design.

—Bob Maksimchuk

I want to express my gratitude to my family for their love and support, which pro vide the foundation for all my endeavors. I wish to thank Grady for giving me the

opportunity to contribute to the third edition of his classic book. Finally, I want to thank Bob Maksimchuk for guiding me in the process of becoming a writer.

—Mike Engle

I would like to dedicate my work to the memory of my mother, Jean Smith, who encouraged me to take on this project. I also want to express my love and appreci ation to my family, Russell, Alyssa, and Logan, for their support and encourage
ment. Thank you, Bob Maksimchuk and Mike Engle, for giving me the opportunity to share in this endeavor.

—Bobbi J. Young

I would like to extend a very special thank you to my husband, Bob, and to my two children, Katherine and Ryan, whose love and support are my true inspiration.

—Kelli A. Houston

Thank you to our reviewers, especially Davyd Norris and Brian Lyons, and to the many other people at Addison-Wesley who worked on this book, especially to Chris Zahn, not only for his development work but also for providing continuity on this long project.

About the Authors

Grady Booch is recognized internationally for his innovative work on software architecture, software engineering, and modeling. He has been with IBM Rational as its Chief Scientist since Rational's founding in 1981. Grady was named an IBM Fellow in March 2003.

Grady is one of the original developers of the Unified Modeling Language (UML) and was also one of the original developers of several of Rational's prod ucts. Grady has served as architect and architectural mentor for numerous com plex software-intensive projects around the world.

Grady is the author of six best-selling books, including the *UML Users Guide* and the seminal *Object-Oriented Analysis with Applications*. Grady has published several hundred technical articles on software engineering, including papers pub lished in the early 1980s that originated the term and practice of object-oriented design. He has lectured and consulted worldwide.

Grady is a member of the Association for Computing Machinery (ACM), the Institute of Electrical and Electronics Engineers (IEEE), the American Associa tion for the Advancement of Science (AAAS), and Computer Professionals for

Social Responsibility (CPSR). He is an IBM Fellow, an ACM Fellow, a World Technology Network Fellow, and a Software Development Forum Visionary. Grady was a founding board member of the Agile Alliance, the Hillside Group, and the Worldwide Institute of Software Architects. He also serves on the advi sory board of Northface University.

Grady received his bachelor of science from the United States Air Force Academy in 1977 and his master of science in electrical engineering from the University of California at Santa Barbara in 1979.

Grady lives with his wife and cats in Colorado. His interests include reading, trav eling, singing, and playing the harp.

Robert A. Maksimchuk is a Research Director in the Unisys Chief Technology Office. He focuses on emerging modeling technologies to advance the strategic direction of the Unisys 3D-Visual Enterprise modeling framework. Bob brings an abundance of systems engineering, modeling, and object-oriented analysis and design expertise, in numerous industries, to this mission. He is the coauthor of the books *UML for Mere Mortals* and *UML for Database Design* and has also written various articles. He has traveled worldwide as a featured speaker in numerous technology forums and led workshops and seminars on UML and object-oriented development. Bob is a member of the Institute of Electrical and Electronics Engi neers (IEEE) and the International Council on Systems Engineering (INCOSE).

Michael W. Engle is a Principal Engineer with the Lockheed Martin Corporation. He has over 26 years of technical and management experience across the com plete system development lifecycle, from project initiation through operations and support. Using his background as a systems engineer, software engineer, and systems architect, Mike employs object-oriented techniques to develop innovative approaches to complex systems development.

Bobbi J. Young, Ph.D., is a Director of Research for the Unisys Chief Technol ogy Office. She has many years of experience in the IT industry working with commercial companies and Department of Defense contractors. Dr. Young has been a consultant mentoring in program management, enterprise architecture, systems engineering, and object-oriented analysis and design. Throughout her career, she has focused on system lifecycle processes and methodologies, as well as enterprise architecture. Dr. Young holds degrees in biology, computer science, and artificial intelligence, and she earned a Ph.D. in management information systems. She is also a Commander (retired) in the United States Naval Reserves.

Jim Conallen is a software engineer in IBM Rational's Model Driven Development Strategy team, where he is actively involved in applying the Object Management Group's (OMG) Model Driven Architecture (MDA) initiative to IBM Rational's model tooling. Jim is also active in the area of asset-based development and the Reusable Asset Specification (RAS). Jim is a frequent conference speaker and article writer. His areas of expertise include Web application development.

He developed the Web Application Extension for UML (WAE), an extension to the UML that lets developers model Web-centric architectures with the UML at appropriate levels of abstraction and detail. This work served as the basis for IBM Rational Rose and Rational XDE Web Modeling functionality.

Jim has authored two editions of the book *Building Web Applications with UML*, the first focusing on Microsoft's Active Server Pages and the latest on J2EE technologies.

Jim's experiences are also drawn from his years prior to Rational, when he was an independent consultant, Peace Corps volunteer, and college instructor, and from his life as a father of three boys. Jim has undergraduate and graduate degrees from Widener University in computer and software engineering.

Kelli Houston is a Consulting IT Specialist at IBM Rational. She is the method architect for IBM's internal method authoring method and is part of the team responsible for integrating IBM's methods. In addition to her method architect role, Kelli also leads the Rational Method Composer (RMC) Special Interest Group (SIG) within IBM and provides consulting and mentoring services to customers and internal IBM consultants on the effective use of RMC.

This page intentionally left blank

Concepts

Sir Isaac Newton secretly admitted to some friends:
He understood how gravity behaved, but not how it
worked!
LILY
TOMLIN
The Search for Signs of Intelligent Life in the Universe

In the early days of object technology, many people were initially introduced to "OO" through programming languages. They discovered what these new languages could do for them and tried to practically apply the languages to solve real-world problems. As time passed, languages improved, development techniques evolved, best practices emerged, and formal object-oriented methodologies were created.

Today object-oriented development is a rich and powerful development model. This section takes a step back to look at the underpinning theory that supplies the foundation for all of the above and provides insight into why things work the way they do in the object-oriented paradigm.

This page intentionally left blank

Complexity

A physician, a civil engineer, and a computer scientist were arguing about what was the oldest profession in the world. The physician remarked, “Well, in the Bible, it says that God created Eve from a rib taken out of Adam. This clearly required surgery, and so I can rightly claim that mine is the oldest profession in the world.” The civil engineer interrupted, and said, “But even earlier in the book of Genesis, it states that God created the order of the heavens and the earth from out of the chaos. This was the first and certainly the most spectacular application of civil engineering. Therefore, fair doctor, you are wrong: mine is the oldest profession in the world.” The computer scientist leaned back in her chair, smiled, and then said confidently, “Ah, but who do you think created the chaos?”

“The more complex the system, the more open it is to total breakdown” [5]. Rarely would a builder think about adding a new sub-basement to an existing 100-story building. Doing that would be very costly and would undoubtedly invite failure. Amazingly, users of software systems rarely think twice about asking for equivalent changes. Besides, they argue, it is only a simple matter of programming.

Our failure to master the complexity of software results in projects that are late, over budget, and deficient in their stated requirements. We often call this condition the software crisis, but frankly, a malady that has carried on this long must be called normal. Sadly, this crisis translates into the
squandering of human resources—a most precious commodity—as well as a considerable loss of opportunities. There are simply not enough good developers around to create all the new software that users need. Further more, a significant number of the development personnel in any given organization must often be dedicated to the maintenance or preservation

of geriatric software. Given the indirect as well as the direct contribution of software to the economic base of most industrialized countries, and con sidering the ways in which software can amplify the powers of the individ ual, it is unacceptable to allow this situation to continue.

1.1 The Structure of Complex Systems

How can we change this dismal picture? Since the underlying problem springs from the inherent complexity of software, our suggestion is to first study how complex systems in other disciplines are organized. Indeed, if we open our eyes to the world about us, we will observe successful systems of significant complex ity. Some of these systems are the works of humanity, such as the Space Shuttle, the England/France tunnel, and large business organizations. Many even more complex systems appear in nature, such as the human circulatory system and the structure of a habanero pepper plant.

The Structure of a Personal Computer

A personal computer is a device of moderate complexity. Most are composed of the same major elements: a central processing unit (CPU), a monitor, a keyboard, and some sort of secondary storage device, usually either a CD or DVD drive and hard disk drive. We may take any one of these parts and further decompose it. For example, a CPU typically encompasses primary memory, an arithmetic/logic unit
(ALU), and a bus to which peripheral devices are attached. Each of these parts may in turn be further decomposed: An ALU may be divided into registers and random control logic, which themselves are constructed from even more primitive elements, such as NAND gates, inverters, and so on.

Here we see the hierarchic nature of a complex system. A personal computer functions properly only because of the collaborative activity of each of its major parts. Together, these separate parts logically form a whole. Indeed, we can rea son about how a computer works only because we can decompose it into parts that we can study separately. Thus, we may study the operation of a monitor inde pendently of the operation of the hard disk drive. Similarly, we may study the ALU without regard for the primary memory subsystem.

Not only are complex systems hierarchic, but the levels of this hierarchy represent different levels of abstraction, each built upon the other, and each understandable by itself. At each level of abstraction, we find a collection of devices that collabo rate to provide services to higher layers. We choose a given level of abstraction to suit our particular needs. For instance, if we were trying to track down a timing

problem in the primary memory, we might properly look at the gate-level archi tecture of the computer, but this level of abstraction would be inappropriate if we were trying to find the source of a problem in a spreadsheet application.

The Structure of Plants and Animals

In botany, scientists seek to understand the similarities and differences among plants through a study of their morphology, that is, their form and structure. Plants are complex multicellular organisms, and from the cooperative activity of various plant organ systems arise such complex behaviors as photosynthesis and transpiration.

Plants consist of three major structures (roots, stems, and leaves). Each of these has a different, specific structure. For example, roots encompass branch roots, root hairs, the root apex, and the root cap. Similarly, a cross-section of a leaf reveals its epidermis, mesophyll, and vascular tissue. Each of these structures is further composed of a collection of cells, and inside each cell we find yet another level of complexity, encompassing such elements as chloroplasts, a nucleus, and so on. As with the structure of a computer, the parts of a plant form a hierarchy, and each level of this hierarchy embodies its own complexity.

All parts at the same level of abstraction interact in well-defined ways. For exam ple, at the highest level of abstraction, roots are responsible for absorbing water and minerals from the soil. Roots interact with stems, which transport these raw materials up to the leaves. The leaves in turn use the water and minerals provided by the stems to produce food through photosynthesis.

There are always clear boundaries between the outside and the inside of a given level. For example, we can state that the parts of a leaf work together to provide the functionality of the leaf as a whole and yet have little or no direct interaction with the elementary parts of the roots. In simpler terms, there is a clear separation of concerns among the parts at different levels of abstraction.

In a computer, we find NAND gates used in the design of the CPU as well as in the hard disk drive. Likewise, a considerable amount of commonality cuts across all parts of the structural hierarchy of a plant. This is God's way of achieving an economy of expression. For example, cells serve as the basic building blocks in all structures of a plant; ultimately, the roots, stems, and leaves of a plant are all composed of cells. Yet, although each of these primitive elements is indeed a cell, there are many different kinds of cells. For example, there are cells with and with out chloroplasts, cells with walls that are impervious to water and cells with walls that are permeable, and even living cells and dead cells.

In studying the morphology of a plant, we do not find individual parts that are each responsible for only one small step in a single larger process, such as photo synthesis. In fact, there are no centralized parts that directly coordinate the activi ties of lower-level ones. Instead, we find separate parts that act as independent agents, each of which exhibits some fairly complex behavior, and each of which contributes to many higher-level functions. Only through the mutual cooperation of meaningful collections of these agents do we see the higher-level functionality of a plant. The science of complexity calls this emergent behavior: The behavior of the whole is greater than the sum of its parts [6].

Turning briefly to the field of zoology, we note that multicellular animals exhibit a hierarchical structure similar to that of plants: Collections of cells form tissues,

tissues work together as organs, clusters of organs define systems (such as the digestive system), and so on. We cannot help but again notice God's awesome economy of expression: The fundamental building block of all animal matter is the cell, just as the cell is the elementary structure of all plant life. Granted, there are differences between these two. For example, plant cells are enclosed by rigid cellulose walls, but animal cells are not. Notwithstanding these differences, how ever, both of these structures are undeniably cells. This is an example of common ality that crosses domains.

A number of mechanisms above the cellular level are also shared by plant and animal life. For example, both use some sort of vascular system to transport nutri ents within the organism, and both exhibit differentiation by sex among members of the same species.

The Structure of Matter

The study of fields as diverse as astronomy and nuclear physics provides us with many other examples of incredibly complex systems. Spanning these two disci plines, we find yet another structural hierarchy. Astronomers study galaxies that are arranged in clusters. Stars, planets, and debris are the constituents of galaxies. Likewise, nuclear physicists are concerned with a structural hierarchy, but one on an entirely different scale. Atoms are made up of electrons, protons, and neutrons; electrons appear to be elementary particles, but protons, neutrons, and other parti cles are formed from more basic components called quarks.

Again we find that a great commonality in the form of shared mechanisms unifies this vast hierarchy. Specifically, there appear to be only four distinct kinds of forces at work in the universe: gravity, electromagnetic interaction, the strong force, and the weak force. Many laws of physics involving these elementary forces, such as the laws of conservation of energy and of momentum, apply to galaxies as well as quarks.

The Structure of Social Institutions

As a final example of complex systems, we turn to the structure of social institu tions. Groups of people join together to accomplish tasks that cannot be done by individuals. Some organizations are transitory, and some endure beyond many lifetimes. As organizations grow larger, we see a distinct hierarchy emerge. Multinational corporations contain companies, which in turn are made up of divi sions, which in turn contain branches, which in turn encompass local offices, and so on. If the organization endures, the boundaries among these parts may change, and over time, a new, more stable hierarchy may emerge.

The relationships among the various parts of a large organization are just like those found among the components of a computer, or a plant, or even a galaxy. Specifically, the degree of interaction among employees within an individual office is greater than that between employees of different offices. A mail clerk usually does not interact with the chief executive officer of a company but does

interact frequently with other people in the mail room. Here, too, these different levels are unified by common mechanisms. The clerk and the executive are both paid by the same financial organization, and both share common facilities, such as the company's telephone system, to accomplish their tasks.

1.2 The Inherent Complexity of Software

A dying star on the verge of collapse, a child learning how to read, white blood cells rushing to attack a virus: These are but a few of the objects in the physical world that involve truly awesome complexity. Software may also involve ele ments of great complexity; however, the complexity we find here is of a funda mentally different kind. As Brooks points out, "Einstein argued that there must be simplified explanations of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer. Much of the complexity that he must master is arbitrary complexity" [1].

Defining Software Complexity

We do realize that some software systems are not complex. These are the largely forgettable applications that are specified, constructed, maintained, and used by the same person, usually the amateur programmer or the professional developer working in isolation. This is not to say that all such systems are crude and inele gant, nor do we mean to belittle their creators. Such systems tend to have a very limited purpose and a very short life span. We can afford to throw them away and

replace them with entirely new software rather than attempt to reuse them, repair them, or extend their functionality. Such applications are generally more tedious than difficult to develop; consequently, learning how to design them does not interest us.

Instead, we are much more interested in the challenges of developing what we will call industrial-strength software. Here we find applications that exhibit a very rich set of behaviors, as, for example, in reactive systems that drive or are driven by events in the physical world, and for which time and space are scarce resources; applications that maintain the integrity of hundreds of thousands of records of information while allowing concurrent updates and queries; and sys tems for the command and control of real-world entities, such as the routing of air or railway traffic. Software systems such as these tend to have a long life span, and over time, many users come to depend on their proper functioning. In the world of industrial-strength software, we also find frameworks that simplify the creation of domain-specific applications, and programs that mimic some aspect of human intelligence. Although such applications are generally products of research and development, they are no less complex, for they are the means and artifacts of incremental and exploratory development.

The distinguishing characteristic of industrial-strength software is that it is intensely difficult, if not impossible, for the individual developer to comprehend all the subtleties of its design. Stated in blunt terms, the complexity of such sys tems exceeds the human intellectual capacity. Alas, this complexity we speak of seems to be an essential property of all large software systems. By *essential* we mean that we may master this complexity, but we can never make it go away.

Why Software Is Inherently Complex

As Brooks suggests, "The complexity of software is an essential property, not an accidental one" [3]. We observe that this inherent complexity derives from four elements: the complexity of the problem domain, the difficulty of managing the development process, the flexibility possible through software, and the problems of characterizing the behavior of discrete systems.

The Complexity of the Problem Domain

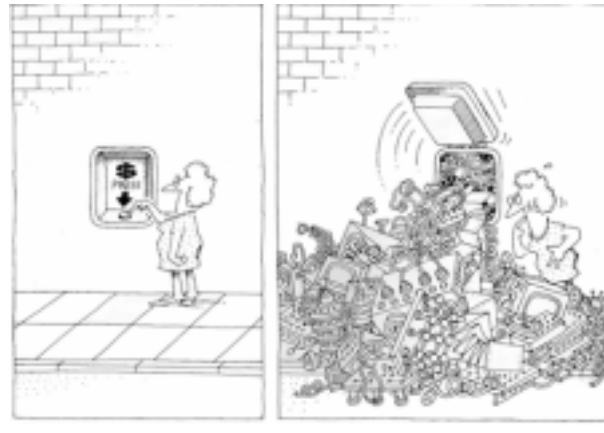
The problems we try to solve in software often involve elements of inescapable complexity, in which we find a myriad of competing, perhaps even contradictory, requirements. Consider the requirements for the electronic system of a multi engine aircraft, a cellular phone switching system, or an autonomous robot. The raw functionality of such systems is difficult enough to comprehend, but now add

all of the (often implicit) nonfunctional requirements such as usability, perfor mance, cost, survivability, and reliability. This unrestrained external complexity is what causes the arbitrary complexity about which Brooks writes.

This external complexity usually springs from the "communication gap" that exists between the users of a system and its developers: Users generally find it very hard to give precise expression to their needs in a form that developers can understand. In some cases, users may have only vague ideas of what they want in a software system. This is not so much the fault of either the users or the develop ers of a system; rather, it occurs because each group generally lacks expertise in the domain of the other. Users and developers have different perspectives on the nature of the problem and make different assumptions regarding the nature of the solution. Actually, even if users had perfect knowledge of their needs, we cur rently have few instruments for precisely capturing these requirements. The com mon way to express requirements is with large volumes of text, occasionally accompanied by a few drawings. Such documents are difficult to comprehend, are open to varying interpretations, and too often contain elements that are designs rather than essential requirements.

A further complication is that the requirements of a software system often change during its development, largely because the very existence of a software develop ment project alters the rules of the problem. Seeing early products, such as design documents and prototypes, and then using a system once it is installed and opera tional are forcing functions that lead users to better understand and

articulate their real needs. At the same time, this process helps developers master the problem
domain, enabling them to ask better questions that illuminate the dark corners of a system's desired behavior.



The task of the software development team is to engineer the illusion of simplicity.

Because a large software system is a capital investment, we cannot afford to scrap an existing system every time its requirements change. Planned or not, systems tend to evolve over time, a condition that is often incorrectly labeled software maintenance. To be more precise, it is *maintenance* when we correct errors; it is *evolution* when we respond to changing requirements; it is *preservation* when we continue to use extraordinary means to keep an ancient and decaying piece of software in operation. Unfortunately, reality suggests that an inordinate percent age of software development resources are spent on software preservation.

The Difficulty of Managing the Development Process

The fundamental task of the software development team is to engineer the illusion of simplicity—to shield users from this vast and often arbitrary external complex ity. Certainly, size is no great virtue in a software system. We strive to write less code by inventing clever and powerful mechanisms that give us this illusion of simplicity, as well as by reusing frameworks of existing designs and code. How ever, the sheer volume of a system's requirements is sometimes inescapable and forces us either to write a large amount of new software or to reuse existing soft ware in novel ways. Just a few decades ago, assembly language programs of only a few thousand lines of code stressed the limits of our software engineering abili ties. Today, it is not unusual to find delivered systems whose size is measured in hundreds of thousands or even millions of lines of code (and all of that in a high order programming language, as well). No one person can ever understand such a system completely. Even if we decompose our implementation in meaningful ways, we still end up with hundreds and sometimes thousands of separate mod ules. This amount of work demands that

we use a team of developers, and ideally we use as small a team as possible. However, no matter what its size, there are always significant challenges associated with team development. Having more developers means more complex communication and hence more difficult coordi nation, particularly if the team is geographically dispersed, as is often the case. With a team of developers, the key management challenge is always to maintain a unity and integrity of design.

The Flexibility Possible through Software

A home-building company generally does not operate its own tree farm from which to harvest trees for lumber; it is highly unusual for a construction firm to build an onsite steel mill to forge custom girders for a new building. Yet in the software industry such practice is common. Software offers the ultimate flexibil ity, so it is possible for a developer to express almost any kind of abstraction. This flexibility turns out to be an incredibly seductive property, however, because it also forces the developer to craft virtually all the primitive building blocks on

which these higher-level abstractions stand. While the construction industry has uniform building codes and standards for the quality of raw materials, few such standards exist in the software industry. As a result, software development remains a labor-intensive business.

The Problems of Characterizing the Behavior of Discrete Systems

If we toss a ball into the air, we can reliably predict its path because we know that under normal conditions, certain laws of physics apply. We would be very surprised if just because we threw the ball a little harder, halfway through its flight it sud denly stopped and shot straight up into the air.[1] In a not-quite-debugged software simulation of this ball's motion, exactly that kind of behavior can easily occur.

Within a large application, there may be hundreds or even thousands of variables as well as more than one thread of control. The entire collection of these vari ables, their current values, and the current address and calling stack of each pro cess within the system constitute the present state of the application. Because we execute our software on digital computers, we have a system with discrete states. By contrast, analog systems such as the motion of the tossed ball are continuous systems. Parnas suggests, "when we say that a system is described by a continu ous function, we are saying that it can contain no hidden surprises. Small changes in inputs will always cause correspondingly small changes in outputs" [4]. On the other hand, discrete systems by their very nature have a finite number of possible states; in large systems, there is a combinatorial explosion that makes this number very large. We try to design our systems with a separation of concerns, so that the behavior in one part of a system has minimal impact on the behavior in another. However, the fact remains that the phase

transitions among discrete states cannot be modeled by continuous functions. Each event external to a software system has the potential of placing that system in a new state, and furthermore, the mapping from state to state is not always deterministic. In the worst circumstances, an external event may corrupt the state of a system because its designers failed to take into account certain interactions among events. When a ship's propulsion

1. Actually, even simple continuous systems can exhibit very complex behavior because of the presence of chaos. Chaos introduces a randomness that makes it impossible to pre cisely predict the future state of a system. For example, given the initial state of two drops of water at the top of a stream, we cannot predict exactly where they will be relative to one another at the bottom of the stream. Chaos has been found in systems as diverse as the weather, chemical reactions, biological systems, and even computer networks. Fortunately, there appears to be underlying order in all chaotic systems, in the form of patterns called attractors.

system fails due to a mathematical overflow, which in turn was caused by some one entering bad data in a maintenance system (a real incident), we understand the seriousness of this issue. There has been a dramatic rise in software-related system failures in subway systems, automobiles, satellites, air traffic control sys tems, inventory systems, and so forth. In continuous systems this kind of behavior would be unlikely, but in discrete systems all external events can affect any part of the system's internal state. Certainly, this is the primary motivation for vigorous testing of our systems, but for all except the most trivial systems, exhaustive test ing is impossible. Since we have neither the mathematical tools nor the intellec tual capacity to model the complete behavior of large discrete systems, we must be content with acceptable levels of confidence regarding their correctness.

1.3 The Five Attributes of a Complex System

Considering the nature of this complexity, we conclude that there are five attributes common to all complex systems.

Hierarchic Structure

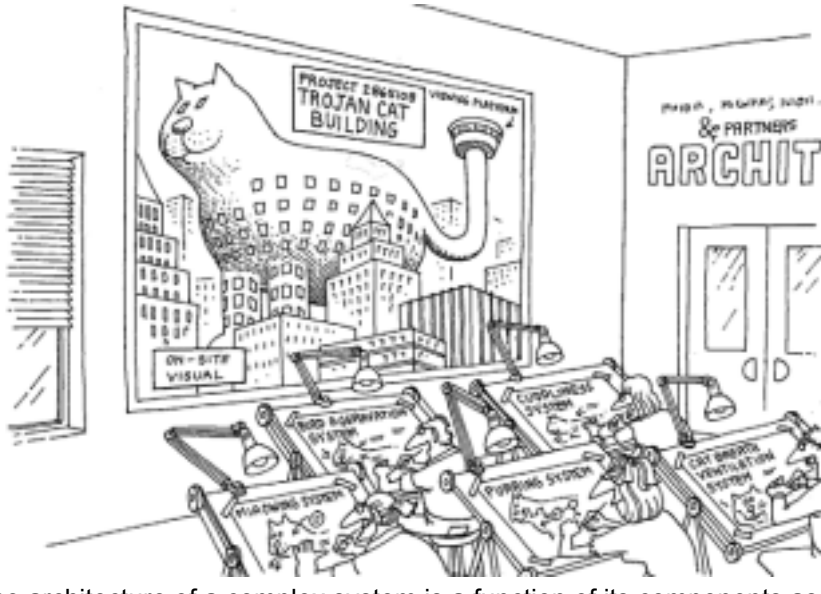
Building on the work of Simon and Ando, Courtois suggests the following:

> Frequently, complexity takes the form of a hierarchy, whereby a complex system is composed of interrelated subsystems that have in turn their own subsystems, and so on, until some lowest level of elementary components is reached. [7]

Simon points out that "the fact that many complex systems have a nearly decom posable, hierarchic structure is a major facilitating factor enabling us to under

stand, describe, and even 'see' such systems and their parts" [8]. Indeed, it is likely that we can understand only those systems that have a hierarchic structure.

It is important to realize that the architecture of a complex system is a function of its components as well as the hierarchic relationships among these components. "All systems have subsystems and all systems are parts of larger systems. . . . The value added by a system must come from the relationships between the parts, not from the parts per se" [9].



The architecture of a complex system is a function of its components as well as the hierarchic relationships among these components.

Relative Primitives

Regarding the nature of the primitive components of a complex system, our expe rience suggests that:

> The choice of what components in a system are primitive is relatively arbitrary and is largely up to the discretion of the observer of the system.

What is primitive for one observer may be at a much higher level of abstraction for another.

Separation of Concerns

Simon calls hierarchic systems *decomposable* because they can be divided into identifiable parts; he calls them *nearly decomposable* because their parts are not completely independent. This leads us to another attribute common to all complex systems:

> Intracomponent linkages are generally stronger than intercomponent linkages. This fact has the effect of separating the high-frequency dynamics of the compo nents—involving the internal structure of the components—from the low frequency dynamics—involving interaction among components. [10]

This difference between intra- and intercomponent interactions provides a clear *separation of concerns* among the various parts of a system, making it possible to study each part in relative isolation.

Common Patterns

As we have discussed, many complex systems are implemented with an economy of expression. Simon thus notes that:

> Hierarchic systems are usually composed of only a few different kinds of sub systems in various combinations and arrangements. [11]

In other words, complex systems have common patterns. These patterns may involve the reuse of small components, such as the cells found in both plants and animals, or of larger structures, such as vascular systems, also found in both plants and animals.

Stable Intermediate Forms

Earlier, we noted that complex systems tend to evolve over time. Specifically, "complex systems will evolve from simple systems much more rapidly if there are stable intermediate forms than if there are not" [12]. In more dramatic terms:

> A complex system that works is invariably found to have evolved from a simple system that worked. . . . A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a working simple system. [13]

As systems evolve, objects that were once considered complex become the primi tive objects on which more complex systems are built. Furthermore, we can never craft these primitive objects correctly the first time: We must use them in context first and then improve them over time as we learn more about the real behavior of the system.

1.4 Organized and Disorganized Complexity

The discovery of common abstractions and mechanisms greatly facilitates our understanding of complex systems. For example, with just a few minutes of orien tation, an experienced pilot can step into a multiengine jet aircraft he or she has

common to all such aircraft, such as the functioning of the rudder, ailerons, and throttle, the pilot primarily needs to learn what properties are unique to that par ticular aircraft. If the pilot already knows how to fly a given aircraft, it is far easier to learn how to fly a similar one.

The Canonical Form of a Complex System

This example suggests that we have been using the term *hierarchy* in a rather loose fashion. Most interesting systems do not embody a single hierarchy; instead, we find that many different hierarchies are usually present within the same complex system. For example, an aircraft may be studied by decomposing it into its propulsion system, flight-control system, and so on. This decomposition represents a structural, or "part of" hierarchy.

Alternately, we can cut across the system in an entirely orthogonal way. For example, a turbofan engine is a specific kind of jet engine, and a Pratt and Whitney TF30 is a specific kind of turbofan engine. Stated another way, a jet engine represents a generalization of the properties common to every kind of jet engine; a turbofan engine is simply a specialized kind of jet engine, with proper ties that distinguish it, for example, from ramjet engines.

This second hierarchy represents an "is a" hierarchy. In our experience, we have found it essential to view a system from both perspectives, studying its "is a" hier archy as well as its "part of" hierarchy. For reasons that will become clear in the next chapter, we call these hierarchies the *class structure* and the *object structure* of the system, respectively.[2]

For those of you who are familiar with object technology, let us be clear. In this case, where we are speaking of class structure and object structure, we are not referring to the classes and objects you create when coding your software. We are referring to classes and objects, at a higher level of abstraction, that make up com plex systems, for example, a jet engine, an airframe, the various types of seats, an autopilot subsystem, and so forth. You will recall from the earlier discussion on the attributes of a complex system that whatever is considered primitive is relative to the observer.

In Figure 1–1 we see the two orthogonal hierarchies of the system: its class struc ture and its object structure. Each hierarchy is layered, with the more abstract

2. Complex software systems embody other kinds of hierarchies as well. Of particular im portance is the module structure, which describes the relationships among the physical components of the system, and the process hierarchy, which describes the relationships among the system's dynamic components.

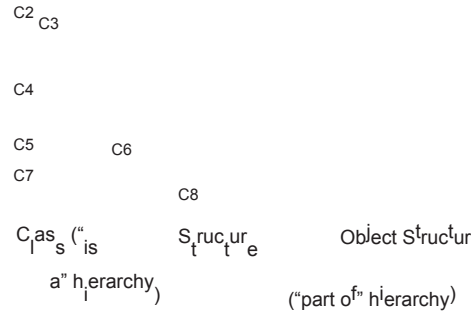


Figure 1–1 The Key Hierarchies of Complex Systems

classes and objects built on more primitive ones. What class or object is chosen as primitive is relative to the problem at hand. Looking inside any given level reveals yet another level of complexity. Especially among the parts of the object struc ture, there are close collaborations among objects at the same level of abstraction.

Combining the concept of the class and object structures together with the five attributes of a complex system (hierarchy, relative primitives [i.e., multiple levels of abstraction], separation of concerns, patterns, and stable intermediate forms), we find that virtually all complex systems take on the same (canonical) form, as we show in Figure 1–2. Collectively, we speak of the class and object structures of a system as its *architecture*.

Notice also that the class structure and the object structure are not completely independent; rather, each object in the object structure represents a specific instance of some class. (In Figure 1–2, note classes C3, C5, C7, and C8 and the number of the instances 03, 05, 07, and 08.) As the figure suggests, there are usu ally many more objects than classes of objects within a complex system. By showing the "part of" as well as the "is a" hierarchy, we explicitly expose the redundancy of the system under consideration. If we did not reveal a system's class structure, we would have to duplicate our knowledge about the properties of each individual part. With the inclusion of the class structure, we capture these common properties in one place.

Also from the same class structure, there are many different ways that these objects can be instantiated and organized. No one particular architecture can really be deemed "correct." This is what makes system architecture challenging— finding the balance between the many ways the components of a system can be structured, the five attributes of complex systems, and the needs of the system user.

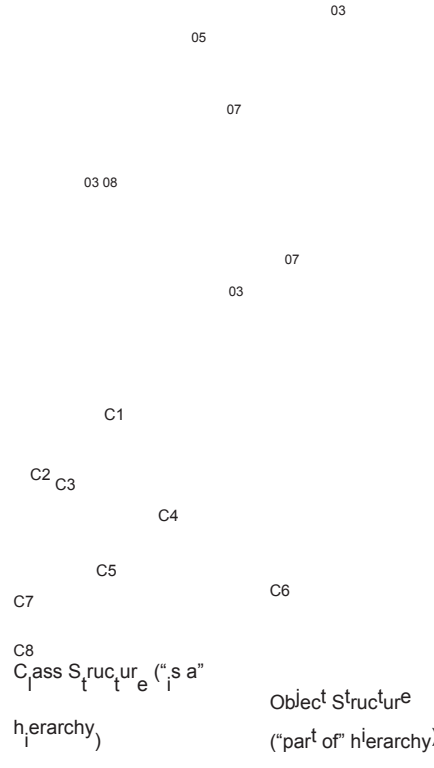


Figure 1–2 The Canonical Form of a Complex System

Our experience is that the most successful complex software systems are those whose designs explicitly encompass well-engineered class and object structures and embody the five attributes of complex systems described in the previous sec tion. Lest the importance of this observation be missed, let us be even more direct: We very rarely encounter software systems that are delivered on time, that are within budget, and that meet their requirements, unless they are designed with these factors in mind.

The Limitations of the Human Capacity for Dealing with Complexity

If we know what the design of complex software systems should be like, then why do we still have serious problems in successfully developing them? This

concept of the organized complexity of software (whose guiding principles we call the *object model*) is relatively new. However, there is yet another factor that dominates: the fundamental limitations of the human capacity for dealing with complexity.

As we first begin to analyze a complex software system, we find many parts that

must interact in a multitude of intricate ways, with little perceptible commonality among either the parts or their interactions; this is an example of disorganized complexity. As we work to bring organization to this complexity through the pro cess of design, we must think about many things at once. For example, in an air traffic control system, we must deal with the state of many different aircraft at once, involving such properties as their location, speed, and heading. Especially in the case of discrete systems, we must cope with a fairly large, intricate, and sometimes nondeterministic state space. Unfortunately, it is absolutely impossible for a single person to keep track of all of these details at once. Experiments by psychologists, such as those of Miller, suggest that the maximum number of chunks of information that an individual can simultaneously comprehend is on the order of seven, plus or minus two [14]. This channel capacity seems to be related to the capacity of short-term memory. Simon additionally notes that pro cessing speed is a limiting factor: It takes the mind about five seconds to accept a new chunk of information [15].

We are thus faced with a fundamental dilemma. The complexity of the software systems we are asked to develop is increasing, yet there are basic limits on our ability to cope with this complexity. How then do we resolve this predicament?

1.5 Bringing Order to Chaos

Certainly, there will always be geniuses among us, people of extraordinary skill who can do the work of a handful of mere mortal developers, the software engi neering equivalents of Frank Lloyd Wright or Leonardo da Vinci. These are the people whom we seek to deploy as our system architects: the ones who devise innovative idioms, mechanisms, and frameworks that others can use as the archi tectural foundations of other applications or systems. However, "The world is only sparsely populated with geniuses. There is no reason to believe that the soft ware engineering community has an inordinately large proportion of them" [2]. Although there is a touch of genius in all of us, in the realm of industrial-strength software we cannot always rely on divine inspiration to carry us through. There fore, we must consider more disciplined ways to master complexity.

The Role of Decomposition

"The technique of mastering complexity has been known since ancient times: divide et impera (divide and rule)" [16]. When designing a complex software sys tem, it is essential to decompose it into smaller and smaller parts, each of which we may then refine independently. In this manner, we satisfy the very real con straint that exists on the channel capacity of human cognition: To understand any given level of a system, we need only comprehend a few parts (rather than all parts) at once. Indeed, as Parnas observes, intelligent decomposition directly addresses the inherent complexity of software by forcing a division of a system's state space [17].

Algorithmic Decomposition

Most of us have been formally trained in the dogma of top-down structured design, and so we approach decomposition as a simple matter of algorithmic decomposition, wherein each module in the system denotes a major step in some overall process. Figure 1–3 is an example of one of the products of structured design, a structure chart that shows the relationships among various functional elements of the solution. This particular structure chart illustrates part of the design of a program that updates the content of a master file. It was automatically generated from a data flow diagram by an expert system tool that embodies the rules of structured design [18].

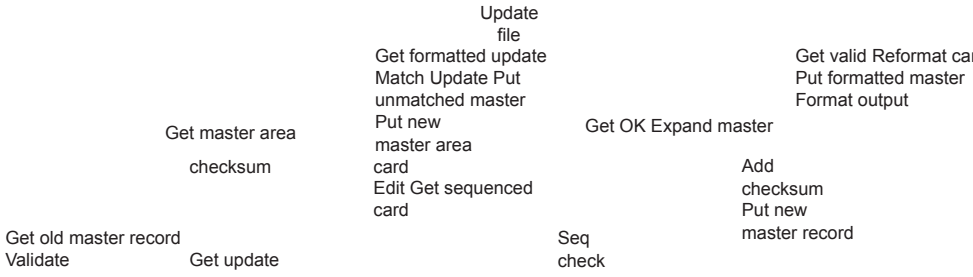


Figure 1–3 Algorithmic Decomposition

Object-Oriented Decomposition

We suggest that there is an alternate decomposition possible for the same prob lem. In Figure 1–4, we have decomposed the system according to the key abstrac tions in the problem domain. Rather than decomposing the problem into steps such as *Get formatted update* and *Add checksum*, we have identified objects such as *Master File* and *Checksum*, which derive directly from the vocabulary of the prob lem domain.

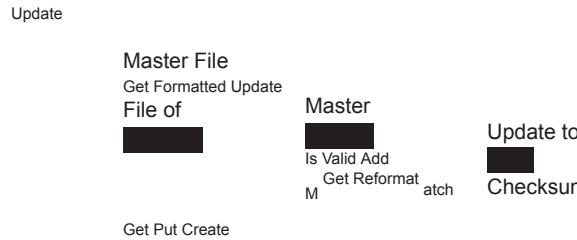


Figure 1–4 Object-Oriented Decomposition

Although both designs solve the same problem, they do so in quite different

ways. In this second decomposition, we view the world as a set of autonomous agents that collaborate to perform some higher-level behavior. *Get Formatted Update* thus does not exist as an independent algorithm; rather, it is an operation associ ated with the object *File of Updates*. Calling this operation creates another object, *Update to Card*. In this manner, each object in our solution embodies its own unique behavior, and each one models some object in the real world. From this perspective, an object is simply a tangible entity that exhibits some well-defined behavior. Objects do things, and we ask them to perform what they do by sending them messages. Because our decomposition is based on objects and not algo rithms, we call this an *object-oriented decomposition*.

Algorithmic versus Object-Oriented Decomposition

Which is the right way to decompose a complex system—by algorithms or by objects? Actually, this is a trick question because the right answer is that both views are important: The algorithmic view highlights the ordering of events, and the object-oriented view emphasizes the agents that either cause action or are the subjects on which these operations act.

Categories of Analysis and Design Methods

We find it useful to distinguish between the terms *method* and *methodology*. A method is a disciplined procedure for generating a set of models that describe various aspects of a software system under development, using some well-defined notation. A methodology is a collection of methods applied across the software development lifecycle and unified by process, practices, and some general, philosophical approach. Methods are impor tant for several reasons. Foremost, they instill a discipline into the develop ment of complex software systems. They define the products that serve as common vehicles for communication among the members of a develop ment team. Additionally, methods define the milestones needed by man agement to measure progress and to manage risk.

Methods have evolved in response to the growing complexity of software systems. In the early days of computing, one simply did not write large pro grams because the capabilities of our machines were greatly limited. The dominant constraints in building systems were then largely due to hard ware: Machines had small amounts of main memory, programs had to con tend with considerable latency within secondary storage devices such as magnetic drums, and processors had cycle times measured in the hun dreds of microseconds. In the 1960s and 1970s the economics of comput ing began to change dramatically as hardware costs plummeted and computer capabilities rose. As a result, it was more desirable and now finally economical to automate more and more applications of increasing complexity. High-order programming languages entered the scene as important tools. Such languages improved the productivity of the individual developer and of the development team as a whole, thus ironically pressur ing us to create software systems of even greater complexity.

Many design methods were proposed during the 1960s and 1970s to address this growing complexity. The most influential of them was top-down structured design, also known as *composite design*. This method was directly influenced by the topology of traditional high-order programming languages, such as FORTRAN and COBOL. In these languages, the fun damental unit of decomposition is the subprogram, and the resulting pro gram takes the shape of a tree in which subprograms perform their work by calling other subprograms. This is exactly the approach taken by top-down structured design: One applies algorithmic decomposition to break a large problem down into smaller steps.

Since the 1960s and 1970s, computers of vastly greater capabilities have evolved. The value of structured design has not changed, but as Stein observes, “Structured programming appears to fall apart when applications exceed 100,000 lines or so of code” [19]. Dozens of design methods have been proposed, many of them invented to deal with the perceived short comings of top-down structured design. The more interesting and successful design methods are cataloged by Peters [20], by Yau and Tsai [21], and in

a comprehensive survey by Teledyne Brown Engineering [22]. Perhaps not surprisingly, many of these methods are largely variations on a similar theme. Indeed, as Sommerville suggests, most methods can be catego rized as one of three kinds [23]:

- Top-down structured design
- Data-driven design
- Object-oriented design

Top-down structured design is exemplified by the work of Yourdon and Constantine [24], Myers [25], and Page-Jones [26]. The foundations of this method derive from the work of Wirth [27, 28] and Dahl, Dijkstra, and Hoare [29]; an important variation on structured design is found in the design method of Mills, Linger, and Hevner [30]. Each of these variations applies algorithmic decomposition. More software has probably been written using these design methods than with any other. Nevertheless, structured design does not address the issues of data abstraction and information hiding, nor does it provide an adequate means of dealing with concurrency. Structured design does not scale up well for extremely complex systems, and this method is largely inappropriate for use with object-based and object-oriented pro gramming languages.

Data-driven design is best exemplified by the early work of Jackson [31, 32] and the methods of Orr [33]. In this method, mapping system inputs to out puts derives the structure of a software system. As with structured design, data-driven design has been successfully applied to a number of complex
domains, particularly information management systems, which involve direct relationships between the inputs and outputs of the system but require little concern for time-critical events.

The underlying concept of object-oriented analysis is that one should

model software systems as collections of cooperating objects, treating individual objects as instances of a class within a hierarchy of classes. Object oriented analysis and design directly reflects the topology of high-order programming languages such as Smalltalk, Object Pascal, C++, the Common Lisp Object System (CLOS), Ada, Eiffel, Python, Visual C#, and Java.

However, the fact remains that we cannot construct a complex system in both ways simultaneously, for they are completely orthogonal views.[3] We must start

3. Langdon suggests that this orthogonality has been studied since ancient times. As he states, "C. H. Waddington has noted that the duality of views can be traced back to the ancient Greeks. A passive view was proposed by Democritus, who asserted that the world was composed of matter called atoms. Democritus' view places things at the center of focus. On the other hand, the classical spokesman for the active view is Heraclitus, who emphasized the notion of process" [34].

decomposing a system either by algorithms or by objects and then use the resulting structure as the framework for expressing the other perspective.

Our experience leads us to apply the object-oriented view first because this approach is better at helping us organize the inherent complexity of software systems, just as it helped us to describe the organized complexity of complex systems as diverse as computers, plants, galaxies, and large social institutions. As we will discuss further in Chapter 2, object-oriented decomposition has a number of highly significant advantages over algorithmic decomposition. Object-oriented decomposition yields smaller systems through the reuse of common mechanisms, thus providing an important economy of expression. Object-oriented systems are also more resilient to change and thus better able to evolve over time because their design is based on stable intermediate forms. Indeed, object-oriented decomposition greatly reduces the risk of building complex software systems because they are designed to evolve incrementally from smaller systems in which we already have confidence. Furthermore, object-oriented decomposition directly addresses the inherent complexity of software by helping us make intelligent decisions regarding the separation of concerns in a large state space.

The Applications section of this book demonstrates these benefits through several applications, drawn from a diverse set of problem domains. The sidebar in this chapter, Categories of Analysis and Design Methods, further compares and contrasts the object-oriented view with more traditional approaches to design.

The Role of Abstraction

Earlier, we referred to Miller's experiments, from which he concluded that an individual can comprehend only about seven, plus or minus two, chunks of information at one time. This number appears to be independent of information content. As Miller himself observes, "The span of absolute judgment and the span of

immediate memory impose severe limitations on the amount of information that we are able to receive, process and remember. By organizing the stimulus input simultaneously into several dimensions and successively into a sequence of chunks, we manage to break . . . this informational bottleneck" [35]. In contempo rary terms, we call this process *chunking* or *abstraction*.

As Wulf describes it, "We (humans) have developed an exceptionally powerful technique for dealing with complexity. We abstract from it. Unable to master the entirety of a complex object, we choose to ignore its inessential details, dealing instead with the generalized, idealized model of the object" [36]. For example, when studying how photosynthesis works in a plant, we can focus on the chemical reactions in certain cells in a leaf and ignore all other parts, such as the roots and stems. We are still constrained by the number of things that we can comprehend

at one time, but through abstraction, we use chunks of information with increas ingly greater semantic content. This is especially true if we take an object-oriented view of the world because objects, as abstractions of entities in the real world, represent a particularly dense and cohesive clustering of information. Chapter 2 examines the meaning of abstraction in much greater detail.

The Role of Hierarchy

Another way to increase the semantic content of individual chunks of information is by explicitly recognizing the class and object hierarchies within a complex soft ware system. The object structure is important because it illustrates how different objects collaborate with one another through patterns of interaction that we call
mechanisms. The class structure is equally important because it highlights com mon structure and behavior within a system. Thus, rather than study each individ ual photosynthesizing cell within a specific plant leaf, it is enough to study one such cell because we expect that all others will exhibit similar behavior. Although we treat each instance of a particular kind of object as distinct, we may assume that it shares the same behavior as all other instances of that same kind of object. By classifying objects into groups of related abstractions (e.g., kinds of plant cells versus animal cells), we come to explicitly distinguish the common and distinct properties of different objects, which further helps us to master their inherent complexity [37].

Identifying the hierarchies within a complex software system is often not easy because it requires the discovery of patterns among many objects, each of which may embody some tremendously complicated behavior. Once we have exposed these hierarchies, however, the structure of a complex system, and in turn our understanding of it, becomes vastly simplified. Chapter 3 considers in detail the nature of class and object hierarchies, and Chapter 4 describes techniques that facilitate our identification of these patterns.

1.6 On Designing Complex Systems

The practice of every engineering discipline—be it civil, mechanical, chemical, electrical, or software engineering—involves elements of both science and art. As Petroski eloquently states, "The conception of a design for a new structure can involve as much a leap of the imagination and as much a synthesis of experience and knowledge as any artist is required to bring to his canvas or paper. And once that design is articulated by the engineer as artist, it must be analyzed by the engi neer as scientist in as rigorous an application of the scientific method as any

scientist must make" [38]. Similarly, Dijkstra observes, "the programming challenge is a large-scale exercise in applied abstraction and thus requires the abilities of the formal mathematician blended with the attitude of the competent engineer" [39].

Engineering as a Science and an Art

The role of the engineer as artist is particularly challenging when the task is to design an entirely new system. Especially in the case of reactive systems and sys tems for command and control, we are frequently asked to write software for an entirely unique set of requirements, often to be executed on a configuration of tar get processors constructed specifically for this system. In other cases, such as the creation of frameworks, tools for research in artificial intelligence, or information management systems, we may have a well-defined, stable target environment, but our requirements may stress the software technology in one or more dimensions. For example, we may be asked to craft systems that are faster, have greater capac ity, or have radically improved functionality. In all these situations, we try to use proven abstractions and mechanisms (the "stable intermediate forms," in Simon's words) as a foundation on which to build new complex systems. In the presence of a large library of reusable software components, the software engineer must assemble these parts in innovative ways to satisfy the stated and implicit require ments, just as the painter or the musician must push the limits of his or her medium.

The Meaning of Design

In every engineering discipline, design encompasses the disciplined approach we use to invent a solution for some problem, thus providing a path from require ments to implementation. In the context of software engineering, Mostow sug gests that the purpose of design is to construct a system that:

- Satisfies a given (perhaps informal) functional specification
- Conforms to limitations of the target medium

- Meets implicit or explicit requirements on performance and resource usage
- Satisfies implicit or explicit design criteria on the form of the artifact
- Satisfies restrictions on the design process itself, such as its length or cost, or the tools available for doing the design [40]

As Stroustrup suggests, "the purpose of design is to create a clean and relatively simple internal structure, sometimes also called an architecture. . . . A design is the end product of the design process" [41]. Design involves balancing a set of

competing requirements. The products of design are models that enable us to rea son about our structures, make trade-offs when requirements conflict, and in gen eral, provide a blueprint for implementation.

The Importance of Model Building

The building of models has a broad acceptance among all engineering disciplines, largely because model building appeals to the principles of decomposition, abstraction, and hierarchy [42]. Each model within a design describes a specific aspect of the system under consideration. As much as possible, we seek to build new models upon old models in which we already have confidence. Models give us the opportunity to fail under controlled conditions. We evaluate each model in both expected and unusual situations, and then we alter them when they fail to behave as we expect or desire.

We have found that in order to express all the subtleties of a complex system, we must use more than one kind of model. For example, when designing a personal computer, an electrical engineer must take into consideration the component-level view of the system as well as the physical layout of the circuit boards. This com ponent view forms a logical picture of the design of the system, which helps the engineer to reason about the cooperative behavior of the components. The board
layout represents the physical packaging of these components, constrained by the board size, available power, and the kinds of components that exist. From this view, the engineer can independently reason about factors such as heat dissipation and manufacturability. The board designer must also consider dynamic as well as static aspects of the system under construction. Thus, the electrical engineer uses diagrams showing the static connections among individual components, as well as timing diagrams that show the behavior of these components over time. The engineer can then employ tools such as oscilloscopes and digital analyzers to val idate the correctness of both the static and dynamic models.

The Elements of Software Design Methodologies

Clearly, there is no magic, no "silver bullet" [43] that can unfailingly lead the software engineer down the path from requirements to the implementation of a complex software system. In fact, the design of complex software systems does

not lend itself at all to cookbook approaches. Rather, as noted earlier in the fifth attribute of complex systems, the design of such systems involves an incremental and iterative process.

Still, sound design methods do bring some much-needed discipline to the devel opment process. The software engineering community has evolved dozens of dif ferent design methodologies, which we can loosely classify into three categories

(see the Categories of Analysis and Design Methods sidebar). Despite their differ ences, all of these have elements in common. Specifically, each includes the following:

- Notation The language for expressing each model
- Process The activities leading to the orderly construction of the system's models
- Tools The artifacts that eliminate the tedium of model building and enforce rules about the models themselves, so that errors and inconsistencies can be exposed

A sound design method is based on a solid theoretical foundation yet offers degrees of freedom for artistic innovation.

The Models of Object-Oriented Development

Is there a "best" design method? No, there is no absolute answer to this question, which is actually just a veiled way of asking the earlier question: What is the best way to decompose a complex system? To reiterate, we have found great value in building models that are focused on the "things" we find in the problem space, forming what we refer to as an object-oriented decomposition.

Object-oriented analysis and design is the method that leads us to an object oriented decomposition. By applying object-oriented design, we create software that is resilient to change and written with economy of expression. We achieve a greater level of confidence in the correctness of our software through an intelli gent separation of its state space. Ultimately, we reduce the risks inherent in developing complex software systems.

In this chapter, we have made a case for using object-oriented analysis and design to master the complexity associated with developing software systems. Addition ally, we have suggested a number of fundamental benefits to be derived from applying this method. Before we present the notation and process of object-ori ented design, however, we must study the principles on which object-oriented development is founded, namely, abstraction, encapsulation, modularity, hierar chy, typing, concurrency, and persistence.

Summary

- Software is inherently complex; the complexity of software systems often exceeds the human intellectual capacity.

- The task of the software development team is to engineer the illusion of simplicity.
- Complexity often takes the form of a hierarchy; it is useful to model both the "is a" and the "part of" hierarchies of a complex system.
- Complex systems generally evolve from stable intermediate forms. ■ There are fundamental limiting factors of human cognition; we can address these constraints through the use of decomposition, abstraction, and hierarchy.
- Complex systems can be viewed by focusing on either things or processes; there are compelling reasons for applying object-oriented decomposition, in which we view the world as a meaningful collection of objects that collabo rate to achieve some higher-level behavior.
- Object-oriented analysis and design is the method that leads us to an object oriented decomposition; object-oriented design uses a notation and process for constructing complex software systems and offers a rich set of models with which we may reason about different aspects of the system under consideration.

The Object Model

Object-oriented technology is built on a sound engineering foundation, whose elements we collectively call the *object model of development* or simply the *object model*. The object model encompasses the principles of abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence. By themselves, none of these principles are new. What is important about the object model is that these elements are brought together in a synergistic way.

Let there be no doubt that object-oriented analysis and design is funda mentally different than traditional structured design approaches: It requires a different way of thinking about decomposition, and it produces software architectures that are largely outside the realm of the structured design culture.

2.1 The Evolution of the Object Model

Object-oriented development did not spontaneously generate itself from the ashes of the uncounted failed software projects that used earlier technologies. It is not a radical departure from earlier approaches. Indeed, it is founded in the best ideas from prior technologies. In this section we will examine the evolution of the tools of our profession to help us understand the foundation and emergence of object oriented technology.

As we look back on the relatively brief yet colorful history of software engineer ing, we cannot help but notice two sweeping trends:

1. The shift in focus from programming-in-the-small to programming-in-the large
2. The evolution of high-order programming languages

Most new industrial-strength software systems are larger and more complex than their predecessors were even just a few years ago. This growth in complexity has prompted a significant amount of useful applied research in software engineering,
particularly with regard to decomposition, abstraction, and hierarchy. The devel opment of more expressive programming languages has complemented these advances.

The Generations of Programming Languages

Wegner has classified some of the more popular high-order programming lan guages in generations arranged according to the language features they first intro duced [2]. (By no means is this an exhaustive list of all programming languages.)

- First-generation languages (1954–1958)
  FORTRAN I Mathematical expressions
  ALGOL 58 Mathematical expressions

Flowmatic Mathematical expressions

IPL V Mathematical expressions

- Second-generation languages (1959–1961)

  FORTRAN II Subroutines, separate compilation

  ALGOL 60 Block structure, data types

  COBOL Data description, file handling

  Lisp List processing, pointers, garbage collection

- Third-generation languages (1962–1970)

  PL/1 FORTRAN + ALGOL + COBOL

  ALGOL 68 Rigorous successor to ALGOL 60

  Pascal Simple successor to ALGOL 60

  Simula Classes, data abstraction

- The generation gap (1970–1980)

  Many different languages were invented, but few endured. However, the fol lowing are worth noting:

  C Efficient; small executables

  FORTRAN 77 ANSI standardization

Let's expand on Wegner's categories.

- Object-orientation boom (1980–1990, but few languages survive)

  Smalltalk 80 Pure object-oriented language

  C++ Derived from C and Simula

  Ada83 Strong typing; heavy Pascal influence

  Eiffel Derived from Ada and Simula

- Emergence of frameworks (1990–today)

  Much language activity, revisions, and standardization have occurred, lead ing to programming frameworks.

  Visual Basic Eased development of the graphical user interface (GUI) for Windows applications

  Java Successor to Oak; designed for portability Python

  Object-oriented scripting language

  J2EE Java-based framework for enterprise computing .NET

  Microsoft's object-based framework

  Visual C# Java competitor for the Microsoft .NET Framework

  Visual Basic .NET Visual Basic for the Microsoft .NET Framework

In successive generations, the kind of abstraction mechanism each language sup ported changed. First-generation languages were used primarily for scientific and engineering applications, and the vocabulary of this problem domain was almost entirely mathematics. Languages such as FORTRAN I were thus developed to allow the programmer to write mathematical formulas, thereby freeing the pro grammer from some of the intricacies of assembly or machine language. This first generation of high-order programming languages therefore represented a step closer to the problem space and a step further away from the underlying machine.

Among second-generation languages, the emphasis was on algorithmic abstrac tions. By this time, machines were becoming more and more powerful, and the economics of the computer industry meant that more kinds of problems could be automated, especially for business applications. Now, the focus was largely on telling the machine what to do: read these personnel records first, sort them next, and then print this report. Again, this new generation of high-order programming languages moved us a step closer to the problem space and further away from the underlying machine.

By the late 1960s, especially with the advent of transistors and then integrated cir cuit technology, the cost of computer hardware had dropped dramatically, yet pro cessing capacity had grown almost exponentially. Larger problems could now be solved, but these demanded the manipulation of more kinds of data. Thus, third generation languages such as ALGOL 60 and, later, Pascal evolved with support

for data abstraction. Now a programmer could describe the meaning of related kinds of data (their type) and let the programming language enforce these design decisions. This generation of high-order programming languages again moved our software a step closer to the problem domain and further away from the underlying machine.

The 1970s provided us with a frenzy of activity in programming language research, resulting in the creation of literally a couple of thousand different pro gramming languages and dialects. To a large extent, the drive to write larger and larger programs highlighted the inadequacies of earlier languages; thus, many new language mechanisms were developed to address these limitations. Few of these languages survived (have you seen a recent textbook on the languages Fred, Chaos, or Tranquil?); however, many of the concepts that they introduced found their way into successors of earlier languages.

What is of the greatest interest to us is the class of languages we call *object-based* and *object-oriented*. Object-based and object-oriented programming languages best support the object-oriented decomposition of software. The number of these languages (and the number of "objectified" variants of existing languages) boomed in the 1980s and early 1990s. Since 1990 a few languages have emerged as mainstream OO languages with the backing of commercial programming tool vendors (e.g., Java, C++). The emergence of programming frameworks (e.g., J2EE, .NET), which provide a tremendous amount of support to the programmer by offering components and services that simplify the common and often mun dane programming tasks, has greatly boosted productivity and demonstrated the elusive promise of component reuse.

The Topology of First- and Early Second Generation Programming Languages

Let's consider the structure of each generation of programming languages. In Fig

ure 2–1, we see the topology of most first- and early second-generation program ming languages. By *topology*, we mean the basic physical building blocks of the language and how those parts can be connected. In this figure, we see that for lan guages such as FORTRAN and COBOL, the basic physical building block of all applications is the subprogram (or the paragraph, for those who speak COBOL).

Applications written in these languages exhibit a relatively flat physical structure, consisting only of global data and subprograms. The arrows in this figure indicate dependencies of the subprograms on various data. During design, one can logi cally separate different kinds of data from one another, but there is little in these languages that can enforce these design decisions. An error in one part of a pro gram can have a devastating ripple effect across the rest of the system because the global data structures are exposed for all subprograms to see.

Figure 2–1 The Topology of First- and Early Second-Generation Programming Languages

When modifications are made to a large system, it is difficult to maintain the integrity of the original design. Often, entropy sets in: After even a short period of maintenance, a program written in one of these languages usually contains a tre mendous amount of cross-coupling among subprograms, implied meanings of data, and twisted flows of control, thus threatening the reliability of the entire sys tem and certainly reducing the overall clarity of the solution.

The Topology of Late Second- and Early Third-Generation Programming Languages

By the mid-1960s, programs were finally being recognized as important interme diate points between the problem and the computer [3]. “The first software abstraction, now called the ‘procedural’ abstraction, grew directly out of this pragmatic view of software. . . . Subprograms were invented prior to 1950, but were not fully appreciated as abstractions at the time. . . . Instead, they were orig inally seen as labor-saving devices. . . . Very quickly though, subprograms were appreciated as a way to abstract program functions” [4].

The realization that subprograms could serve as an abstraction mechanism had three important consequences. First, languages were invented that supported a variety of parameter-passing mechanisms. Second, the foundations of structured programming were laid, manifesting themselves in language support for the nest ing of subprograms and the development of theories regarding control structures and the scope and visibility of declarations. Third, structured design methods emerged, offering guidance to designers trying to build large systems using sub programs as basic physical building blocks. Thus, it is not surprising, as Figure 2–2 shows, that the topology of late second- and early third-generation languages is largely a variation on the theme of earlier generations. This topology addresses

Figure 2–2 The Topology of Late Second- and Early Third-Generation Programming Languages

some of the inadequacies of earlier languages, namely, the need to have greater control over algorithmic abstractions, but it still fails to address the problems of programming-in-the-large and data design.

The Topology of Late Third-Generation Programming Languages

Starting with FORTRAN II, and appearing in most late third-generation program languages, another important structuring mechanism evolved to address the grow ing issues of programming-in-the-large. Larger programming projects meant larger development teams, and thus the need to develop different parts of the same program independently. The answer to this need was the separately compiled module, which in its early conception was little more than an arbitrary container for data and subprograms, as Figure 2–3 shows. Modules were rarely recognized as an important abstraction mechanism; in practice they were used simply to group subprograms that were most likely to change together.

Most languages of this generation, while supporting some sort of modular struc ture, had few rules that required semantic consistency among module interfaces. A developer writing a subprogram for one module might assume that it would be

called with three different parameters: a floating-point number, an array of ten elements, and an integer representing a Boolean flag. In another module, a call to this subprogram might incorrectly use actual parameters that violated these assumptions: an integer, an array of five elements, and a negative number. Simi larly, one module might use a block of common data that it assumed as its own, and another module might violate these assumptions by directly manipulating this

Figure 2–3 The Topology of Late Third-Generation Programming Languages

data. Unfortunately, because most of these languages had dismal support for data abstraction and strong typing, such errors could be detected only during execution of the program.

The Topology of Object-Based and Object Oriented Programming Languages

Data abstraction is important to mastering complexity. “The nature of abstrac tions that may be achieved through the use of procedures is well suited to the description of abstract operations, but is not particularly well suited to the description of abstract objects. This is a serious drawback, for in many applica tions, the complexity of the data objects to be manipulated contributes substan tially to the overall complexity of the problem” [5]. This realization had two important consequences. First, data-driven design methods emerged, which pro vided a disciplined approach to the problems of doing data abstraction in algorith mically oriented languages. Second, theories regarding the concept of a type appeared, which eventually found their realization in languages such as Pascal.

The natural conclusion of these ideas first appeared in the language Simula and was improved upon, resulting in the development of several languages such as Smalltalk, Object Pascal, C++, Ada, Eiffel, and Java. For reasons that we will explain shortly, these languages are called object-based or object-oriented. Figure 2–4 illustrates the topology of such languages for small to moderate-sized applications.

Figure 2–4 The Topology of Small to Moderate-Sized Applications Using Object-Based and Object-Oriented Programming Languages

The physical building block in such languages is the module, which represents a logical collection of classes and objects instead of subprograms, as in earlier lan guages. To state it another way, "If procedures and functions are verbs and pieces of data are nouns, a procedure-oriented program is organized around verbs while an object-oriented program is organized around nouns" [6]. For this reason, the physical structure of a small to moderate-sized object-oriented application appears as a graph, not as a tree, which is typical of algorithmically oriented lan guages. Additionally, there is little or no global data. Instead, data and operations are united in such a way that the fundamental logical building blocks of our sys tems are no longer algorithms, but instead are classes and objects.

By now we have progressed beyond programming-in-the-large and must cope with programming-in-the-colossal. For very complex systems, we find that classes, objects, and modules provide an essential yet insufficient means of abstraction. Fortunately, the object model scales up. In large systems, we find clusters of abstractions built in layers on top of one another. At any given level of abstraction, we find meaningful collections of objects that collaborate to achieve some higher-level behavior. If we look inside any given cluster to view its imple mentation, we unveil yet another set of cooperative abstractions. This is exactly the organization of complexity described in Chapter 1; this topology is shown in Figure 2–5.

Figure 2–5 The Topology of Large Applications Using Object-Based and Object-Oriented Programming Languages

2.2 Foundations of the Object Model

Structured design methods evolved to guide developers who were trying to build complex systems using algorithms as their fundamental building blocks. Simi larly, object-oriented design methods have evolved to help developers exploit the expressive power of object-based and object-oriented programming languages, using the class and object as basic building blocks.

Actually, the object model has been influenced by a number of factors, not just object-oriented programming. Indeed, as further discussed in the sidebar, Founda tions—The Object Model, the object model has proven to be a unifying concept in computer science, applicable not just to programming languages but also to the design of user interfaces, databases, and even computer architectures. The reason for this widespread appeal is simply that an object orientation helps us to cope with the complexity inherent in many different kinds of systems.

Object-oriented analysis and design thus represents an evolutionary development, not a revolutionary one; it does not break with advances from the past but builds on proven ones. Unfortunately, most programmers are not

rigorously trained in OOAD. Certainly, many good engineers have developed and deployed countless useful software systems using structured design techniques. However, there are limits to the amount of complexity we can handle using only algorithmic decom position; thus we must turn to object-oriented decomposition. Furthermore, if we try to use languages such as C++ and Java as if they were only traditional, algo rithmically oriented languages, we not only miss the power available to us, but we usually end up worse off than if we had used an older language such as C or Pascal. Give a power drill to a carpenter who knows nothing about electricity, and he would use it as a hammer. He will end up bending quite a few nails and smash ing several fingers, for a power drill makes a lousy hammer.

Because the object model derives from so many disparate sources, it has unfortu nately been accompanied by a muddle of terminology. A Smalltalk programmer uses *methods*, a C++ programmer uses *virtual member functions*, and a CLOS programmer uses *generic functions*. An Object Pascal programmer talks of a *type coercion*; an Ada programmer calls the same thing a *type conversion*; a C# or Java programmer would use a *cast*. To minimize the confusion, let's define what is object-oriented and what is not.

The phrase *object-oriented* "has been bandied about with carefree abandon with much the same reverence accorded 'motherhood,' 'apple pie,' and 'structured pro gramming'"[7]. What we can agree on is that the concept of an object is central to anything object-oriented. In the previous chapter, we informally defined an object as a tangible entity that exhibits some well-defined behavior. Stefik and Bobrow
define objects as "entities that combine the properties of procedures and data since they perform computations and save local state" [8]. Defining objects as entities begs the question somewhat, but the basic concept here is that objects serve to unify the ideas of algorithmic and data abstraction. Jones further clarifies
this term by noting that "in the object model, emphasis is placed on crisply char acterizing the components of the physical or abstract system to be modeled by a programmed system. . . . Objects have a certain 'integrity' which should not—in fact, cannot—be violated. An object can only change state, behave, be manipu lated, or stand in relation to other objects in ways appropriate to that object. Stated differently, there exist invariant properties that characterize an object and its behavior. An elevator, for example, is characterized by invariant properties including [that] it only travels up and down inside its shaft. . . . Any elevator sim ulation must incorporate these invariants, for they are integral to the notion of an elevator" [32].

Foundations—The Object Model

As Yonezawa and Tokoro point out, "The term 'object' emerged almost independently in various fields in computer science, almost simultaneously in the early 1970s, to refer to notions that were different in their appear ance, yet mutually related. All of these notions were invented to manage the complexity of software systems in such a way that objects represented

components of a modularly decomposed system or modular units of knowl edge representation" [9]. Levy adds that the following events have contrib uted to the evolution of object-oriented concepts:

- Advances in computer architecture, including capability systems and hard ware support for operating systems concepts
- Advances in programming languages, as demonstrated in Simula, Smalltalk, CLU, and Ada
- Advances in programming methodology, including modularization and infor mation hiding [10]

We would add to this list three more contributions to the foundation of the object model:

- Advances in database models
- Research in artificial intelligence
- Advances in philosophy and cognitive science

The concept of an object had its beginnings in hardware over twenty years ago, starting with the invention of descriptor-based architectures and, later, capability-based architectures [11]. These architectures represented a break from the classical von Neumann architectures and came about through attempts to close the gap between the high-level abstractions of programming languages and the low-level abstractions of the machine itself [12]. According to its proponents, the advantages of such architec tures are many: better error detection, improved execution efficiency, fewer instruction types, simpler compilation, and reduced storage requirements. Computers can also have an object-oriented architecture.

Closely related to developments in object-oriented architectures are object-oriented operating systems. Dijkstra's work with the THE multipro gramming system first introduced the concept of building systems as lay ered state machines [18]. Other pioneering object-oriented operating systems include the Plessey/System 250 (for the Plessey 250 multiproces sor), Hydra (for CMU's C.mmp), CALTSS (for the CDC 6400), CAP (for the Cambridge CAP computer), UCLA Secure UNIX (for the PDP 11/45 and 11/70), StarOS (for CMU's Cm*), Medusa (also for CMU's Cm*), and iMAX (for the Intel 432) [19].

Perhaps the most important contribution to the object model derives from the class of programming languages we call object-based and object oriented. The fundamental ideas of classes and objects first appeared in

the language Simula 67. The Flex system, followed by various dialects of Smalltalk, such as Smalltalk-72, -74, and -76, and finally the current ver sion, Smalltalk-80, took Simula's object-oriented paradigm to its natural conclusion by making everything in the language an instance of a class. In the 1970s languages such as Alphard, CLU, Euclid, Gypsy, Mesa, and Modula were developed, which supported the then-emerging ideas of data abstraction. Language research led to the grafting of Simula and Smalltalk concepts onto traditional high-order programming languages. The unifica tion of object-oriented concepts with C has lead to the languages C++ and Objective C. Then Java arrived to help programmers avoid common pro gramming errors often seen when using C++. Adding object-oriented pro

gramming mechanisms to Pascal has led to the languages Object Pascal, Eiffel, and Ada. Additionally, many dialects of Lisp incorporate the object oriented features of Simula and Smalltalk. Appendix A discusses some of these and other programming language developments in greater detail.

The first person to formally identify the importance of composing systems in layers of abstraction was Dijkstra. Parnas later introduced the idea of information hiding [20], and in the 1970s a number of researchers, most notably Liskov and Zilles [21], Guttag [22], and Shaw [23], pioneered the development of abstract data type mechanisms. Hoare contributed to these developments with his proposal for a theory of types and subclasses [24].

Although database technology has evolved somewhat independently of software engineering, it has also contributed to the object model [25], pri marily through the ideas of the entity-relationship (ER) approach to data modeling [26]. In the ER model, first proposed by Chen [27], the world is modeled in terms of its entities, the attributes of these entities, and the rela tionships among these entities.

In the field of artificial intelligence, developments in knowledge representa tion have contributed to an understanding of object-oriented abstractions. In 1975, Minsky first proposed a theory of frames to represent real-world objects as perceived by image and natural language recognition systems [28]. Since then, frames have been used as the architectural foundation for a variety of intelligent systems.

Lastly, philosophy and cognitive science have contributed to the advance ment of the object model. The idea that the world could be viewed in terms of either objects or processes was a Greek innovation, and in the seven teenth century, we find Descartes observing that humans naturally apply an object-oriented view of the world [29]. In the twentieth century, Rand expanded on these themes in her philosophy of objectivist epistemology [30]. More recently, Minsky has proposed a model of human intelligence in which he considers the mind to be organized as a society of otherwise mindless agents [31]. Minsky argues that only through the cooperative behavior of these agents do we find what we call *intelligence*.

Object-Oriented Programming

What, then, is object-oriented programming (OOP)? We define it as follows:

> Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

There are three important parts to this definition: (1) Object-oriented program ming uses objects, not algorithms, as its fundamental logical building blocks (the "part of" hierarchy we introduced in Chapter 1); (2) each object is an instance of some class; and (3) classes may be related to one another via inheritance relation ships (the "is a" hierarchy we spoke of in Chapter 1). A program may appear to be object-oriented, but if any of these elements is missing, it is not an

object-oriented program. Specifically, programming without inheritance is distinctly not object oriented; that would merely be programming with abstract data types.

By this definition, some languages are object-oriented, and some are not. Strous trup suggests that "if the term 'object-oriented language' means anything, it must mean a language that has mechanisms that support the object-oriented style of programming well. . . . A language supports a programming style well if it pro vides facilities that make it convenient to use that style. A language does not sup port a technique if it takes exceptional effort or skill to write such programs; in that case, the language merely enables programmers to use the techniques" [33]. From a theoretical perspective, one can fake object-oriented programming in non object-oriented programming languages like Pascal and even COBOL or assem bly language, but it is horribly ungainly to do so. Cardelli and Wegner thus say:

> [A] language is object-oriented if and only if it satisfies the following requirements:
>
> - It supports objects that are data abstractions with an interface of named operations and a hidden local state.
> - Objects have an associated type [class].
> - Types [classes] may inherit attributes from supertypes [superclasses]. [34]

For a language to support inheritance means that it is possible to express "is a" relationships among types, for example, a red rose is a kind of flower, and a flower is a kind of plant. If a language does not provide direct support for inherit ance, then it is not object-oriented. Cardelli and Wegner distinguish such lan guages by calling them *object-based* rather than *object-oriented*. Under this definition, Smalltalk, Object Pascal, C++, Eiffel, CLOS, C#, and Java are all object-oriented, and Ada83 is object-based (support for object orientation was later added to Ada95). However, since objects and classes are elements of both

kinds of languages, it is both possible and highly desirable for us to use object oriented design methods for both object-based and object-oriented programming languages.

Object-Oriented Design

The emphasis in programming methods is primarily on the proper and effective use of particular language mechanisms. By contrast, design methods emphasize the proper and effective structuring of a complex system. What, then, is object oriented design (OOD)? We suggest the following:

> Object-oriented design is a method of design encompassing the process of object oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design.

There are two important parts to this definition: object-oriented design (1) leads to an object-oriented decomposition and (2) uses different notations to express different models of the logical (class and object structure) and physical (module

and process architecture) design of a system, in addition to the static and dynamic aspects of the system.

The support for object-oriented decomposition is what makes object-oriented design quite different from structured design: The former uses class and object abstractions to logically structure systems, and the latter uses algorithmic abstrac tions. We will use the term *object-oriented design* to refer to any method that leads to an object-oriented decomposition.

Object-Oriented Analysis

The object model has influenced even earlier phases of the software development lifecycle. Traditional structured analysis techniques, best typified by the work of DeMarco [35], Yourdon [36], and Gane and Sarson [37], with real-time exten sions by Ward and Mellor [38] and by Hatley and Pirbhai [39], focus on the flow of data within a system. Object-oriented analysis (OOA) emphasizes the building of real-world models, using an object-oriented view of the world:

> Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.

How are OOA, OOD, and OOP related? Basically, the products of object-oriented analysis serve as the models from which we may start an object-oriented design;

the products of object-oriented design can then be used as blueprints for com pletely implementing a system using object-oriented programming methods.

2.3 Elements of the Object Model

Jenkins and Glasgow observe that "most programmers work in one language and use only one programming style. They program in a paradigm enforced by the language they use. Frequently, they have not been exposed to alternate ways of thinking about a problem, and hence have difficulty in seeing the advantage of choosing a style more appropriate to the problem at hand" [40]. Bobrow and Stefik define a programming style as "a way of organizing programs on the basis of some conceptual model of programming and an appropriate language to make programs written in the style clear" [41]. They further suggest that there are five main kinds of programming styles, listed here with the kinds of abstractions they employ:

1. Procedure-oriented Algorithms
2. Object-oriented Classes and objects
3. Logic-oriented Goals, often expressed in a predicate calculus 4.

Rule-oriented If–then rules

5. Constraint-oriented Invariant relationships

There is no single programming style that is best for all kinds of applications. For example, rule-oriented programming would be best suited for the design of a knowledge base, and procedure-oriented programming would be best for the design of computation-intense operations. From our experience, the object-oriented style is best suited to the broadest set of applications; indeed, this programming paradigm often serves as the architectural framework in which we employ other paradigms.

Each of these styles of programming is based on its own conceptual framework. Each requires a different mindset, a different way of thinking about the problem. For all things object-oriented, the conceptual framework is the object model. There are four major elements of this model:

1. Abstraction
2. Encapsulation
3. Modularity
4. Hierarchy

By *major*, we mean that a model without any one of these elements is not object oriented.

There are three minor elements of the object model:

1. Typing
2. Concurrency
3. Persistence

By *minor*, we mean that each of these elements is a useful, but not essential, part of the object model.

Without this conceptual framework, you may be programming in a language such as Smalltalk, Object Pascal, C++, Eiffel, or Ada, but your design is going to smell like a FORTRAN, Pascal, or C application. You will have missed out on or other wise abused the expressive power of the object-oriented language you are using
for implementation. More importantly, you are not likely to have mastered the complexity of the problem at hand.

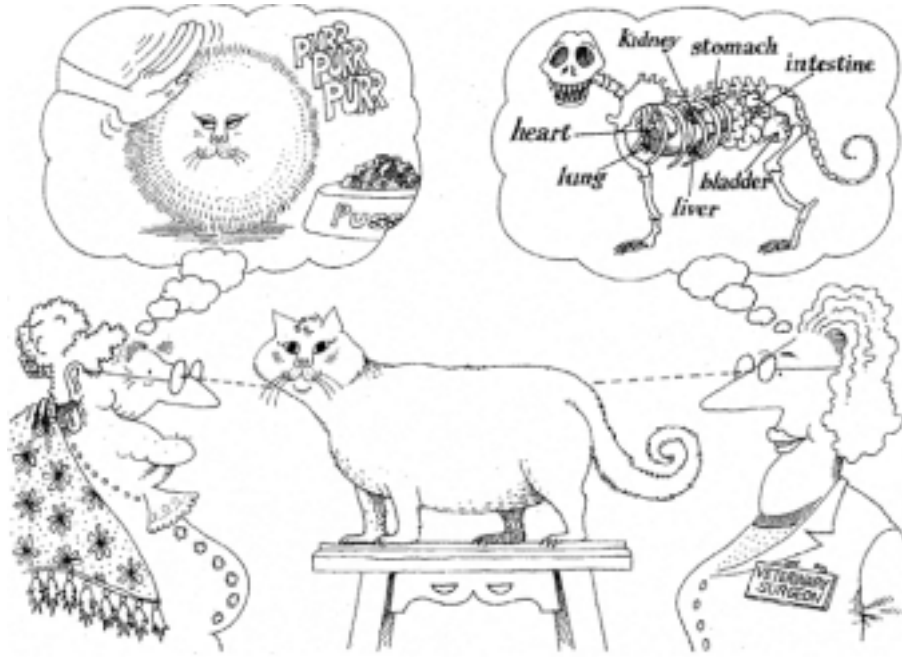
The Meaning of Abstraction

Abstraction is one of the fundamental ways that we as humans cope with com plexity. Dahl, Dijkstra, and Hoare suggest that “abstraction arises from a recogni tion of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon these similarities and to ignore for the time being the differences” [42]. Shaw defines an abstraction as “a simplified description, or specification, of a system that emphasizes some of the system’s

details or properties while suppressing others. A good abstraction is one that emphasizes details that are significant to the reader or user and suppresses details that are, at least for the moment, immaterial or diversionary" [43]. Berzins, Gray, and Naumann recommend that "a concept qualifies as an abstraction only if it can be described, understood, and analyzed independently of the mechanism that will eventually be used to realize it" [44]. Combining these different viewpoints, we define an abstraction as follows:

> An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.

An abstraction focuses on the outside view of an object and so serves to separate an object's essential behavior from its implementation. Abelson and Sussman call this behavior/implementation division an abstraction barrier [45] achieved by applying the *principle of least commitment*, through which the interface of an object provides its essential behavior, and nothing more [46]. We like to use an additional principle that we call the *principle of least astonishment*, through which an abstraction captures the entire behavior of some object, no more and no less, and offers no surprises or side effects that go beyond the scope of the abstraction.



Abstraction focuses on the essential characteristics of some object, relative to the perspective of the viewer.

Deciding on the right set of abstractions for a given domain is the central problem in object-oriented design. Because this topic is so important, the whole of Chapter 4 is devoted to it.

"There is a spectrum of abstraction, from objects which closely model problem

domain entities to objects which really have no reason for existence" [47]. From the most to the least useful, these kinds of abstractions include the following:

- Entity abstraction An object that represents a useful model of a problem domain or solution domain entity
- Action abstraction An object that provides a generalized set of operations, all of which perform the same kind of function
- Virtual machine abstraction An object that groups operations that are all used by some superior level of control, or operations that all use some junior-level set of operations
- Coincidental abstraction An object that packages a set of operations that have no relation to each other

We strive to build entity abstractions because they directly parallel the vocabulary of a given problem domain.

A client is any object that uses the resources of another object (known as the server). We can characterize the behavior of an object by considering the services that it provides to other objects, as well as the operations that it may perform on other objects. This view forces us to concentrate on the outside view of an object and leads us to what Meyer calls the *contract model* of programming [48]: the outside view of each object defines a contract on which other objects may depend, and which in turn must be carried out by the inside view of the object itself (often in collaboration with other objects). This contract thus establishes all the assump tions a client object may make about the behavior of a server object. In other words, this contract encompasses the responsibilities of an object, namely, the behavior for which it is held accountable [49].

Individually, each operation that contributes to this contract has a unique signa ture comprising all of its formal arguments and return type. We call the entire set of operations that a client may perform on an object, together with the legal order ings in which they may be invoked, its *protocol*. A protocol denotes the ways in which an object may act and react and thus constitutes the entire static and dynamic outside view of the abstraction.

Central to the idea of an abstraction is the concept of invariance. An *invariant* is some Boolean (true or false) condition whose truth must be preserved. For each operation associated with an object, we may define *preconditions* (invariants assumed by the operation) as well as *postconditions* (invariants satisfied by the operation). Violating an invariant breaks the contract associated with an abstrac tion. If a precondition is violated, this means that a client has not satisfied its part of the bargain, and hence the server cannot proceed reliably. Similarly, if a post condition is violated, this means that a server has not carried out its part of the contract, and so its clients can no longer trust the behavior of the server. An exception is an indication that some invariant has not been or cannot be satisfied. Certain languages permit objects to throw exceptions so as to abandon processing and alert some other object to the problem, which in turn may catch

the exception and handle the problem.

As an aside, the terms *operation*, *method*, and *member function* evolved from three different programming cultures (Ada, Smalltalk, and C++, respectively). They all mean virtually the same thing, so we will use them interchangeably.

All abstractions have static as well as dynamic properties. For example, a file object takes up a certain amount of space on a particular memory device; it has a name, and it has contents. These are all static properties. The value of each of these properties is dynamic, relative to the lifetime of the object: A file object may grow or shrink in size, its name may change, its contents may change. In a

procedure-oriented style of programming, the activity that changes the dynamic value of objects is the central part of all programs; things happen when subpro grams are called and statements are executed. In a rule-oriented style of program ming, things happen when new events cause rules to fire, which in turn may trigger other rules, and so on. In an object-oriented style of programming, things happen whenever we operate on an object (i.e., when we send a message to an object). Thus, invoking an operation on an object elicits some reaction from the object. What operations we can meaningfully perform on an object and how that object reacts constitute the entire behavior of the object.

Examples of Abstraction

Let's illustrate these concepts with some examples. We defer a complete treat ment of how to find the right abstractions for a given problem to Chapter 4.

On a hydroponics farm, plants are grown in a nutrient solution, without sand, gravel, or other soils. Maintaining the proper greenhouse environment is a deli cate job and depends on the kind of plant being grown and its age. One must control diverse factors such as temperature, humidity, light, pH, and nutrient con centrations. On a large farm, it is not unusual to have an automated system that constantly monitors and adjusts these elements. Simply stated, the purpose of an automated gardener is to efficiently carry out, with minimal human intervention, growing plans for the healthy production of multiple crops.

One of the key abstractions in this problem is that of a sensor. Actually, there are several different kinds of sensors. Anything that affects production must be mea sured, so we must have sensors for air and water temperature, humidity, light, pH, and nutrient concentrations, among other things. Viewed from the outside, a tem perature sensor is simply an object that knows how to measure the temperature at some specific location. What is a temperature? It is some numeric value, within a limited range of values and with a certain precision, that represents degrees in the scale of Fahrenheit, Centigrade, or Kelvin, whichever is most appropriate for our problem. What is a location? It is some identifiable place on the farm at which we
desire to measure the temperature; presumably, there are only a few such loca tions. What is important for a temperature sensor is not so much where it is located but the fact that it has a location and identity unique from all other tem

perature sensors. Now we are ready to ask: What are the responsibilities of a tem perature sensor? Our design decision is that a sensor is responsible for knowing the temperature at a given location and reporting that temperature when asked. More concretely, what operations can a client perform on a temperature sensor? Our design decision is that a client can calibrate it, as well as ask what the current temperature is. (See Figure 2–6. Note that this representation is similar to the rep resentation of a class in UML 2.0. You will learn the actual representation in Chapter 5.)

Abstraction: Temperature Sensor

Important Characteristics:

temperature
location

Responsibilities:

report current temperature
calibrate

Figure 2–6 Abstraction of a `Temperature Sensor`

The abstraction we have described thus far is passive; some client object must operate on an air `Temperature Sensor` object to determine its current tem perature. However, there is another legitimate abstraction that may be more or less appropriate depending on the broader system design decisions we might make. Specifically, rather than the `Temperature Sensor` being passive, we might make it active, so that it is not acted on but rather acts on other objects whenever the temperature at its location changes a certain number of degrees from a given setpoint. This abstraction is almost the same as our first one, except that its responsibilities have changed slightly: A sensor is now responsible for reporting the current temperature when it changes, not just when asked. What new operations must this abstraction provide?

This abstraction is a bit more complicated than the first (see Figure 2–7). A client of this abstraction may invoke an operation to establish a critical range of temper atures. It is then the responsibility of the sensor to report whenever the tempera ture at its location drops below or rises above the given setpoint. When the function is invoked, the sensor provides its location and the current temperature, so that the client has sufficient information to respond to the condition.

Abstraction: Active Temperature Sensor

Important Characteristics:

temperature
location
setpoint

Responsibilities:

report current temperature
calibrate

Figure 2–7 Abstraction of an `Active Temperature Sensor`

How the `Active Temperature Sensor` carries out its responsibilities is a function of its inside view and is of no concern to outside clients. These then are the secrets of the class, which are implemented by the class's private parts together with the definition of its member functions.

Let's consider a different abstraction. For each crop, there must be a growing plan that describes how temperature, light, nutrients, and other conditions should change over time to maximize the harvest. A growing plan is a legitimate entity abstrac tion because it forms part of the vocabulary of the problem domain. Each crop has its own growing plan, but the growing plans for all crops take the same form.

A growing plan is responsible for keeping track of all interesting actions associ ated with growing a crop, correlated with the times at which those actions should take place. For example, on day 15 in the lifetime of a certain crop, our growing
plan might be to maintain a temperature of 78°F for 16 hours, turn on the lights for 14 of these hours, and then drop the temperature to 65°F for the rest of the day. We might also want to add certain extra nutrients in the middle of the day,
while still maintaining a slightly acidic pH. From the perspective outside of each growing-plan object, a client must be able to establish the details of a plan, modify a plan, and inquire about a plan, as shown in Figure 2–8. (Note that abstractions are likely to evolve over the lifetime of a project. As details begin to be fleshed out, a responsibility such as "establish plan" could turn into multiple responsibilities, such as "set temperature," "set pH," and so forth. This is to be expected as more knowledge of client requirements is gained, designs mature, and implementation approaches are considered.)

Our decision is also that we will not require a growing plan to carry out its plan: We will leave this as the responsibility of a different abstraction (e.g., a `Plan Controller`). In this manner, we create a clear *separation of concerns* among the logically different parts of the system, so as to reduce the conceptual size of each individual abstraction. For example, there might be an object that sits at the

Abstraction: Growing Plan

Important Characteristics:

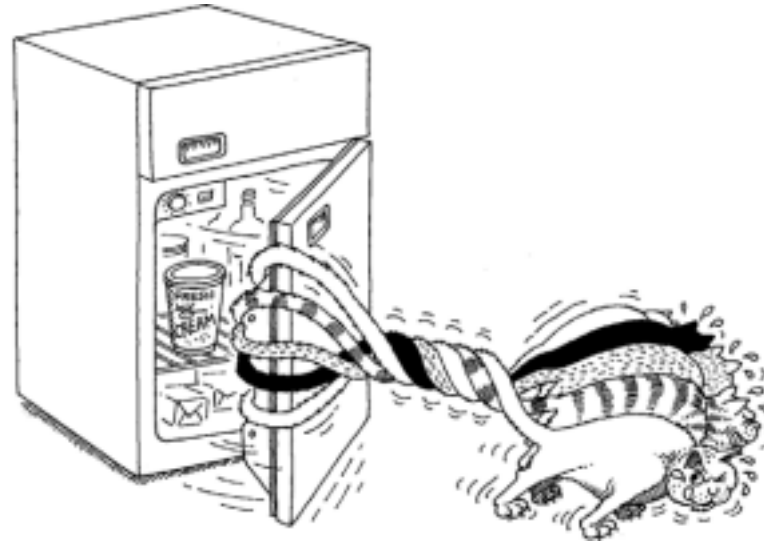
name

Responsibilities:

establish plan
modify plan
clear plan

boundary of the human/machine interface and translates human input into plans. This is the object that establishes the details of a growing plan, so it must be able to change the state of a `Growing Plan` object. There must also be an object that carries out the growing plan, and it must be able to read the details of a plan for a particular time.

As this example points out, no object stands alone; every object collaborates with other objects to achieve some behavior.[1] Our design decisions about how these objects cooperate with one another define the boundaries of each abstraction and thus the responsibilities and protocol of each object.

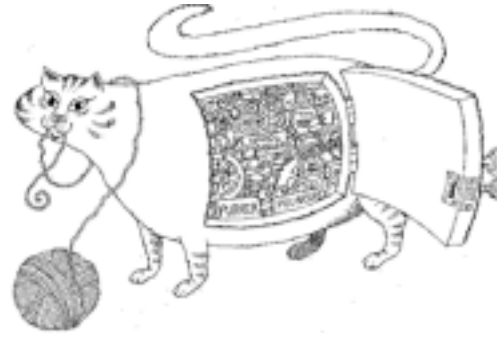


Objects collaborate with other objects to achieve some behavior.

The Meaning of Encapsulation

Although we earlier described our abstraction of the `Growing Plan` as a time/ action mapping, its implementation is not necessarily a literal table or map data structure. Indeed, whichever representation is chosen is immaterial to the client's
contract with the `Growing Plan`, as long as that representation upholds the contract. Simply stated, the abstraction of an object should precede the decisions about its implementation. Once an implementation is selected, it should be treated as a secret of the abstraction and hidden from most clients.

1. Stated another way, with apologies to the poet John Donne, no object is an island (al though an island may be abstracted as an object).



Encapsulation hides the details of the implementation of an object.

Abstraction and encapsulation are complementary concepts: Abstraction focuses on the observable behavior of an object, whereas encapsulation focuses on the implementation that gives rise to this behavior. Encapsulation is most often achieved through information hiding (not just data hiding), which is the process of hiding all the secrets of an object that do not contribute to its essential charac teristics; typically, the structure of an object is hidden, as well as the implementa tion of its methods. "No part of a complex system should depend on the internal details of any other part" [50]. Whereas abstraction "helps people to think about what they are doing," encapsulation "allows program changes to be reliably made with limited effort" [51].

Encapsulation provides explicit barriers among different abstractions and thus leads to a clear separation of concerns. For example, consider again the structure of a plant. To understand how photosynthesis works at a high level of abstraction, we can ignore details such as the responsibilities of plant roots or the chemistry of cell walls. Similarly, in designing a database application, it is standard practice to write programs so that they don't care about the physical representation of data but depend only on a schema that denotes the data's logical view [52]. In both of these cases, objects at one level of abstraction are shielded from implementation details at lower levels of abstraction.

"For abstraction to work, implementations must be encapsulated" [53]. In prac tice, this means that each class must have two parts: an interface and an imple mentation. The interface of a class captures only its outside view, encompassing our abstraction of the behavior common to all instances of the class. The imple mentation of a class comprises the representation of the abstraction as well as the mechanisms that achieve the desired behavior. The interface of a class is the one place where we assert all of the assumptions that a client may make about any instances of the class; the implementation encapsulates details about which no client may make assumptions.

To summarize, we define encapsulation as follows:

> Encapsulation is the process of compartmentalizing the elements of an abstrac

tion that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation.

Britton and Parnas call these encapsulated elements the "secrets" of an abstrac tion [54].

Examples of Encapsulation

To illustrate the principle of encapsulation, let's return to the problem of the Hydroponics Gardening System. Another key abstraction in this problem domain is that of a heater. A heater is at a fairly low level of abstraction, and thus we might decide that there are only three meaningful operations that we can perform on this object: turn it on, turn it off, and find out if it is running.

Separation of Concerns

We do not make it a responsibility of the `Heater` abstraction to maintain a fixed temperature. Instead, we choose to give this responsibility to another object (e.g., the `Heater Controller`), which must collaborate with a temperature sensor and a heater to achieve this higher-level behavior. We call this behavior *higher-level* because it builds on the primitive semantics of temperature sensors and heaters and adds some new semantics, namely, *hysteresis*, which prevents the heater from being turned on and off too rapidly when the temperature is near boundary conditions. By deciding on this separation of responsibilities, we make each individual abstraction more cohesive.

All a client needs to know about the class `Heater` is its available interface (i.e., the responsibilities that it may execute at the client's request—see Figure 2–9).

Turning to the inside view of the `Heater`, we have an entirely different perspec tive. Suppose that our system engineers have decided to locate the computers that control each greenhouse away from the building (perhaps to avoid the harsh envi ronment) and to connect each computer to its sensors and actuators via serial lines. One reasonable implementation for the `Heater` class might be to use an electromechanical relay that controls the power going to each physical heater, with the relays in turn commanded by messages sent along these serial lines. For example, to turn on a heater, we might transmit a special command string, fol lowed by a number identifying the specific heater, followed by another number used to signal turning the heater on.

Abstraction: Heater **Important Characteristics:**

location

status

Responsibilities:

turn on
turn off
provide status

Related Candidate Abstractions: Heater Controller, Temperature Sensor**Figure 2–9** Abstraction of a `Heater`

Suppose that for whatever reason our system engineers choose to use memory mapped I/O instead of serial communication lines. We would not need to change the interface of the `Heater`, yet the implementation would be very different. The client would not see any change at all as the client sees only the `Heater` inter face. This is the key point of encapsulation. In fact, the client should not care what the implementation is, as long as it receives the service it needs from the `Heater`.

Let's next consider the implementation of the class `GrowingPlan`. As we men tioned earlier, a growing plan is essentially a time/action mapping. Perhaps the most reasonable representation for this abstraction would be a dictionary of time/ action pairs, using an open hash table. We need not store an action for every hour, because things don't change that quickly. Rather, we can store actions only for when they change, and have the implementation extrapolate between times.

In this manner, our implementation encapsulates two secrets: the use of an open hash table (which is distinctly a part of the vocabulary of the solution domain, not the problem domain) and the use of extrapolation to reduce our storage require ments (otherwise we would have to store many more time/action pairs over the duration of a growing season). No client of this abstraction need ever know about these implementation decisions because they do not materially affect the out wardly observable behavior of the class.

Intelligent encapsulation localizes design decisions that are likely to change. As a system evolves, its developers might discover that, in actual use, certain opera tions take longer than is acceptable or that some objects consume more space than is available. In such situations, the representation of an object is often changed so that more efficient algorithms can be applied or so that one can optimize for space by calculating rather than storing certain data. This ability to change the represen tation of an abstraction without disturbing any of its clients is the essential benefit of encapsulation.

Hiding is a relative concept: What is hidden at one level of abstraction may repre sent the outside view at another level of abstraction. The underlying representa

tion of an object can be revealed, but in most cases only if the creator of the abstraction explicitly exposes the implementation, and then only if the client is willing to accept the resulting additional complexity. Thus, encapsulation cannot stop a developer from doing stupid things; as Stroustrup points out, "Hiding is for the prevention of accidents, not the prevention of fraud" [56]. Of course, no pro gramming language prevents a human from literally seeing the implementation of a class, although an operating system might deny access to a particular file that contains the implementation of a class.

The Meaning of Modularity

"The act of partitioning a program into individual components can reduce its complexity to some degree. . . . Although partitioning a program is helpful for this reason, a more powerful justification for partitioning a program is that it creates a number of well-defined, documented boundaries within the program. These boundaries, or interfaces, are invaluable in the comprehension of the program" [57]. In some languages, such as Smalltalk, there is no concept of a module, so the class forms the only physical unit of decomposition. Java has packages that contain classes. In many other languages, including Object Pascal, C++, and Ada, the module is a separate language construct and therefore warrants a separate set of design decisions. In these languages, classes and objects form the logical structure of a system; we place these abstractions in modules to produce the system's phys ical architecture. Especially for larger applications, in which we may have many hundreds of classes, the use of modules is essential to help manage complexity.

Modularity packages abstractions into discrete units.

"Modularization consists of dividing a program into modules which can be com piled separately, but which have connections with other modules. We will use the definition of Parnas: 'The connections between modules are the assumptions which the modules make about each other'" [58]. Most languages that support the module as a separate concept also distinguish between the interface of a

module and its implementation. Thus, it is fair to say that modularity and encapsulation go hand in hand.

Deciding on the right set of modules for a given problem is almost as hard a prob lem as deciding on the right set of abstractions. Zelkowitz is absolutely right when he states that “because the solution may not be known when the design stage starts, decomposition into smaller modules may be quite difficult. For older applications (such as compiler writing), this process may become standard, but for new ones (such as defense systems or spacecraft control), it may be quite dif ficult” [59].

Modules serve as the physical containers in which we declare the classes and objects of our logical design. This is no different than the situation faced by the electrical engineer designing a computer motherboard. NAND, NOR, and NOT gates might be used to construct the necessary logic, but these gates must be physically packaged in standard integrated circuits. Lacking any such standard software parts, the software engineer has considerably more degrees of freedom— as if the electrical engineer had a silicon foundry at his or her disposal.

For tiny problems, the developer might decide to declare every class and object in the same package. For anything but the most trivial software, a better solution is to group logically related classes and objects in the same module and to expose only those elements that other modules absolutely must see. This kind of modu larization is a good thing, but it can be taken to extremes. For example, consider an application that runs on a distributed set of processors and uses a message passing mechanism to coordinate the activities of different programs. In a large system, such as a command and control system, it is common to have several hun dred or even a few thousand kinds of messages. A naive strategy might be to define each message class in its own module. As it turns out, this is a singularly poor design decision. Not only does it create a documentation nightmare, but it makes it terribly difficult for any users to find the classes they need. Furthermore, when decisions change, hundreds of modules must be modified or recompiled. This example shows how information hiding can backfire [60]. Arbitrary modu larization is sometimes worse than no modularization at all.

In traditional structured design, modularization is primarily concerned with the meaningful grouping of subprograms, using the criteria of coupling and cohesion. In object-oriented design, the problem is subtly different: The task is to decide where to physically package the classes and objects, which are distinctly different from subprograms.