Matt Weisfeld

Fifth Edition

The Object-Oriented Thought Process

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

The Object-Oriented Thought Process

Fifth Edition

The Object-Oriented Thought Process

Fifth Edition

Matt Weisfeld

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi
Mexico City • São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2019930825

Copyright © 2019 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearsoned.com/permissions/.

ISBN-13: 978-0-13-518196-6

ISBN-10: 0-13-518196-8

1 19

Microsoft and/or its respective suppliers make no representations about the suitability of the information contained in the documents and related graphics published as part of the services for any purpose. All such documents and related graphics are provided “as is” without warranty of any kind. Microsoft and/ or its respective suppliers hereby disclaim all warranties and conditions with regard to this information, including all warranties and conditions of merchantability, whether express, implied or statutory, fitness for a particular purpose, title and non-infringement. In no event shall Microsoft and/or its respective sup-pliers be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of information available from the services.

The documents and related graphics contained herein could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein. Microsoft and/or its respective sup-pliers may make improvements and/or changes in the product(s) and/or the program(s) described herein at any time. Partial screenshots may be viewed in full within the software version specified.

Microsoft® and Windows® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. Screenshots and icons reprinted with permission from the Microsoft Corporation. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

Editor-in-Chief
Mark Taub

Development Editor
Mark Taber

Managing Editor
Sandra Schroeder

Senior Project Editor
Tonya Simpson

Indexer
Erika Millen

Proofreader
Abigail Manheim

Technical Reviewer
John Upchurch

Editorial Assistant
Cindy Teeters

Cover Designer
Chuti Prasertsith

Compositor
codeMantra

❖

To Sharon, Stacy, Stephanie, and Paulo

❖

Contents at a Glance

Introduction 1

1 Introduction to Object-Oriented Concepts 5

2 How to Think in Terms of Objects 33

3 More Object-Oriented Concepts 47

4 The Anatomy of a Class 67

5 Class Design Guidelines 77

6 Designing with Objects 91

7 Mastering Inheritance and Composition 105

8 Frameworks and Reuse: Designing with Interfaces and Abstract Classes 125

9 Building Objects and Object-Oriented Design 147

10 Design Patterns 161

11 Avoiding Dependencies and Highly Coupled Classes 175

12 The SOLID Principles of Object-Oriented Design 187

Index 205

Table of Contents

Introduction 1
This Book's Scope 1
What's New in the Fifth Edition 2
The Intended Audience 3
The Book's Approach 4
Source Code Used in This Book 4

1 Introduction to Object-Oriented Concepts 5
The Fundamental Concepts 5
Objects and Legacy Systems 6
Procedural Versus OO Programming 7
Moving from Procedural to Object-Oriented Development 11
- Procedural Programming 11
- OO Programming 11

What Exactly Is an Object? 12
- Object Data 12
- Object Behaviors 13

What Exactly Is a Class? 16
- Creating Objects 17
- Attributes 18
- Methods 19
- Messages 19

Using Class Diagrams as a Visual Tool 19
Encapsulation and Data Hiding 20
- Interfaces 20
- Implementations 21
- A Real-World Example of the Interface/Implementation Paradigm 21
- A Model of the Interface/Implementation Paradigm 22

Inheritance 23
- Superclasses and Subclasses 24
- Abstraction 25
- Is-a Relationships 26

Polymorphism 27

Composition 30
- Abstraction 30
- Has-a Relationships 31

Conclusion 31

2 How to Think in Terms of Objects 33

Knowing the Difference Between the Interface and the Implementation 34
- The Interface 36
- The Implementation 36
- An Interface/Implementation Example 36

Using Abstract Thinking When Designing Interfaces 41

Providing the Absolute Minimal User Interface Possible 42
- Determining the Users 43
- Object Behavior 44
- Environmental Constraints 44
- Identifying the Public Interfaces 44
- Identifying the Implementation 45

Conclusion 46

References 46

3 More Object-Oriented Concepts 47

Constructors 47
- When Is a Constructor Called? 48
- What's Inside a Constructor? 48
- The Default Constructor 48
- Using Multiple Constructors 49
- The Design of Constructors 53

Error Handling 54
- Ignoring the Problem 54
- Checking for Problems and Aborting the Application 54
- Checking for Problems and Attempting to Recover 54
- Throwing an Exception 55

The Importance of Scope 57
- Local Attributes 58
- Object Attributes 59
- Class Attributes 61

Operator Overloading 62

Multiple Inheritance 63
Object Operations 63
Conclusion 65
References 65

4 The Anatomy of a Class 67
The Name of the Class 67
Comments 69
Attributes 69
Constructors 71
Accessors 73
Public Interface Methods 75
Private Implementation Methods 76
Conclusion 76
References 76

5 Class Design Guidelines 77
Modeling Real-World Systems 77
Identifying the Public Interfaces 78
The Minimum Public Interface 78
Hiding the Implementation 79
Designing Robust Constructors (and Perhaps Destructors) 80
Designing Error Handling into a Class 81
Documenting a Class and Using Comments 81
Building Objects with the Intent to Cooperate 82
Designing with Reuse in Mind 82
Designing with Extensibility in Mind 83
Making Names Descriptive 83
Abstracting Out Nonportable Code 84
Providing a Way to Copy and Compare Objects 84
Keeping the Scope as Small as Possible 84
Designing with Maintainability in Mind 86
Using Iteration in the Development Process 86
Testing the Interface 86
Using Object Persistence 88
Serializing and Marshaling Objects 89
Conclusion 90
References 90

6 Designing with Objects 91
Design Guidelines 91
Performing the Proper Analysis 95
Developing a Statement of Work 95
Gathering the Requirements 95
Developing a System Prototype 96
Identifying the Classes 96
Determining the Responsibilities of Each Class 96
Determining How the Classes Collaborate with Each Other 96
Creating a Class Model to Describe the System 96
Prototyping the User Interface in Code 97
Object Wrappers 97
Structured Code 98
Wrapping Structured Code 99
Wrapping Nonportable Code 101
Wrapping Existing Classes 101
Conclusion 102
References 103

7 Mastering Inheritance and Composition 105
Reusing Objects 105
Inheritance 106
Generalization and Specialization 109
Design Decisions 110
Composition 112
Representing Composition with UML 113
Why Encapsulation Is Fundamental to OO 115
How Inheritance Weakens Encapsulation 115
A Detailed Example of Polymorphism 117
Object Responsibility 118
Abstract Classes, Virtual Methods, and Protocols 121
Conclusion 123
References 123

8 Frameworks and Reuse: Designing with Interfaces and Abstract Classes 125
Code: To Reuse or Not to Reuse? 125
What Is a Framework? 126

What Is a Contract? 128
Abstract Classes 128
Interfaces 131
Tying It All Together 133
The Compiler Proof 135
Making a Contract 136
System Plug-in Points 138
An E-Business Example 139
An E-Business Problem 139
The Non-Reuse Approach 139
An E-Business Solution 141
The UML Object Model 142
Conclusion 146
References 146

9 Building Objects and Object-Oriented Design 147
Composition Relationships 148
Building in Phases 149
Types of Composition 151
Aggregations 151
Associations 152
Using Associations and Aggregations Together 153
Avoiding Dependencies 154
Cardinality 155
Multiple Object Associations 157
Optional Associations 158
Tying It All Together: An Example 159
Conclusion 160
References 160

10 Design Patterns 161
Why Design Patterns? 162
Smalltalk's Model/View/Controller 163
Types of Design Patterns 164
Creational Patterns 165
Structural Patterns 169
Behavioral Patterns 171

Antipatterns 173
Conclusion 174
References 174

11 Avoiding Dependencies and Highly Coupled Classes 175
Composition versus Inheritance and Dependency Injection 177
1) Inheritance 177
2) Composition 179
Dependency Injection 182
Conclusion 185
References 185

12 The SOLID Principles of Object-Oriented Design 187
The SOLID Principles of Object-Oriented Design 188
1) SRP: Single Responsibility Principle 188
2) OCP: Open/Close Principle 192
3) LSP: Liskov Substitution Principle 194
4) IPS: Interface Segregation Principle 197
5) DIP: Dependency Inversion Principle 198
Conclusion 204
References 204

Index 205

Acknowledgments

As with the first four editions, this book required the combined efforts of many people. I would like to take the time to acknowledge as many of these people as possible, for without them, this book would never have happened.

First and foremost, I would like to thank my wife Sharon for all her help. Not only did she provide support and encouragement throughout this lengthy process, she is also the first line editor for all my writing.

I would also like to thank my mom and the rest of my family for their continued support.

It is hard to believe that the work on the first edition of this book began in 1998. For all these years, I have thoroughly enjoyed working with everyone at Pearson—on all five editions. Working with editors Mark Taber and Tonya Simpson on this edition has been a pleasure.

A special thanks goes to Jon Upchurch for his expertise with much of the code as well as the technical editing of the manuscript. Jon's insights into an amazing range of technical topics have been of great help to me.

Finally, thanks to my daughters, Stacy and Stephanie, and my cat, Paulo, for always keeping me on my toes.

About the Author

Matt Weisfeld is a college professor, software developer, and author based in Cleveland, Ohio. Prior to teaching college full time, he spent 20 years in the information technology industry as a software developer, entrepreneur, and adjunct professor. Weisfeld holds an MS in computer science and an MBA. Besides several editions of *The Object-Oriented Thought Process*, Matt has authored two other software development books and published many articles in magazines and journals, such as *informit.com*, *developer.com*, *Dr. Dobb's Journal*, *The C/C++ Users Journal*, *Software Development Magazine*, *Java Report*, and the international journal *Project Management*.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

We welcome your comments. You can email or write to let us know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that we cannot help you with technical problems related to the topic of this book.

When you write, please be sure to include this book's title and author as well as your name and email address. We will carefully review your comments and share them with the author and editors who worked on the book.

Email: community@informit.com

Reader Services

Visit our website and register this book at www.informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

Figure Credits

Cover image © SOMRERK WITTHAYANANT/Shutterstock.

Figure 8.1, screenshot of Microsoft word copyright © Microsoft 2019.

Figure 8.2, screenshot of API documentation copyright © 1993, 2018, Oracle.

2
How to Think in Terms of Objects

In Chapter 1, "Introduction to Object-Oriented Concepts," you learned the fundamental object-oriented (OO) concepts. The rest of the book delves more deeply into these concepts and introduces several others. Many factors go into a good design, whether it is an OO design or not. The fundamental unit of OO design is the class. The desired end result of OO design is a robust and functional object model—in other words, a complete system.

As with most things in life, there is no single right or wrong way to approach a problem. There are usually many ways to tackle the same problem. So when attempting to design an OO solution, don't get hung up in trying to do a perfect design the first time (there will always be room for improvement). What you really need to do is brainstorm and let your thought process go in different directions. Do not try to conform to any standards or conventions when trying to solve a problem because the whole idea is to be creative.

In fact, at the start of the process, don't even begin to consider a specific programming language. The first order of business is to identify and solve business problems. Work on the conceptual analysis and design first. Think about specific technologies only when they are fundamental to the business problem. For example, you can't design a wireless network without wireless technology. However, it is often the case that you will have more than one software solution to consider.

Thus, before you start to design a system, or even a class, think the problem through and have some fun! In this chapter we explore the fine art and science of OO thinking.

Any fundamental change in thinking is not trivial. As a case in point, a lot has been mentioned about the move from structured to OO development. As was mentioned earlier, one side effect of this debate is the misconception that structured and object-oriented development are mutually exclusive. This is not the case. As we know from our discussion on wrappers, structured and object-oriented development coexist. In fact, when you write an OO application, you are using structured constructs everywhere. I have never seen a program, OO or otherwise, that does not use loops, if-statements, and so on. Yet making the switch to OO design does require a different type of investment.

Changing from FORTRAN to COBOL, or even to C, requires you to learn a new language; however, making the move from COBOL to C++, C# .NET, Visual Basic .NET, Objective-C, Swift, or Java requires you to learn a new thought process. This is where the overused phrase *OO paradigm* rears its ugly head. When moving to an OO language, you must first go through the investment of learning OO concepts and the corresponding thought process. If this paradigm shift does not take place, one of two things will happen: Either the project will not truly be OO in nature (for example, it will use C++ without using OO constructs) or the project will be a complete object-disoriented mess.

Three important things you can do to develop a good sense of the OO thought process are covered in this chapter:

- Knowing the difference between the interface and implementation
- Thinking more abstractly
- Giving the user the minimal interface possible

We have already touched on some of these concepts in Chapter 1, “Introduction to Object-Oriented Concepts,” and we now go into much more detail.

Knowing the Difference Between the Interface and the Implementation

As we saw in Chapter 1, one of the keys to building a strong OO design is to understand the difference between the interface and the implementation. Thus, when designing a class, what the user needs to know and, perhaps of more importance, what the user does not need to know are of vital importance. The data hiding mechanism inherent with encapsulation is the means by which nonessential data is hidden from the user.

> **Caution**
>
> Do not confuse the concept of the interface with terms like *graphical user interface (GUI).* Although a GUI is, as its name implies, an interface, the term *interfaces,* as used here, is more general in nature and is not restricted to a graphical interface.

Remember the toaster example in Chapter 1? The toaster, or any appliance for that matter, is plugged into the interface, which is the electrical outlet—see Figure 2.1. All appliances gain access to the required electricity by complying with the correct interface: the electrical outlet. The toaster doesn’t need to know anything about the implementation or how the electricity is produced. For all the toaster cares, a coal plant or a nuclear plant could produce the electricity—the appliance does not care which, as long as the interface works as specified, correctly and safely.

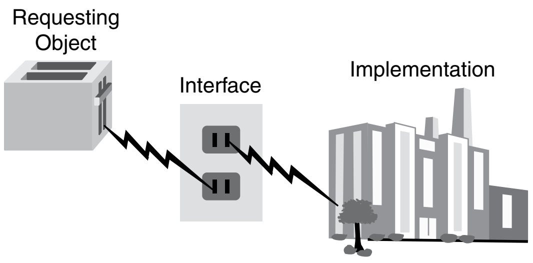

Figure 2.1 Power plant revisited.

As another example, consider an automobile. The interface between you and the car includes components such as the steering wheel, gas pedal, brake, and ignition switch. For most people, aesthetic issues aside, the main concern when driving a car is that the car starts, accelerates, stops, steers, and so on. The implementation, basically the stuff that you don't see, is of little concern to the average driver. In fact, most people would not even be able to identify certain components, such as the catalytic converter and gasket. However, any driver would recognize and know how to use the steering wheel because this is a common interface. By installing a standard steering wheel in the car, manufacturers are assured that the people in their target market will be able to use the system.

If, however, a manufacturer decided to install a joystick in place of the steering wheel, most drivers would balk at this, and the automobile might not be a big seller (except possibly gamers). On the other hand, as long as the performance and aesthetics didn't change, the average driver would not notice whether the manufacturer changed the engine (part of the implementation) of the automobile.

It must be stressed that the interchangeable engines must be identical in every way—as far as the interface goes. Replacing a four-cylinder engine with an eight-cylinder engine would change the rules and likely would not work with other components that interface with the engine, just as changing the current from AC to DC would affect the rules in the power plant example.

The engine is part of the implementation, and the steering wheel is part of the interface. A change in the implementation should have no impact on the driver, whereas a change to the interface might. The driver would notice an aesthetic change to the steering wheel, even if it performs in a similar manner. It must be stressed that a change to the engine that *is* noticeable by the driver breaks this rule. For example, a change that would result in noticeable loss of power is actually impacting the interface.

What Users See

When we talk about users in this chapter, we primarily mean designers and developers—not necessarily end users. Thus, when we talk about interfaces in this context, we are talking about class interfaces, not GUIs.

Properly constructed classes are designed in two parts—the interface and the implementation.

The Interface

The services presented to an end user constitute the interface. In the best case, *only* the services the end user needs are presented. Of course, which services the user needs might be a matter of opinion. If you put 10 people in a room and ask each of them to do an independent design, you might receive 10 totally different designs—and there is nothing wrong with that. However, as a general rule, the interface to a class should contain only what the user needs to know. In the toaster example, the user needs to know only that the toaster must be plugged into the interface (which in this case is the electrical outlet) and how to operate the toaster itself.

> **Identifying the User**
>
> Perhaps the most important consideration when designing a class is identifying the audience, or users, of the class.

The Implementation

The implementation details are hidden from the user. One goal regarding the implementation should be kept in mind: A change to the implementation *should not* require a change to the user's code. This might seem a bit confusing, but this goal is at the heart of the design issue.

> **Good Interfaces**
>
> If the interface is designed properly, a change to the implementation should not require a change to the user's code.

Remember that the interface includes the syntax to call a method and return a value. If this interface does not change, the user does not care whether the implementation is changed. As long as the programmer can use the same syntax and retrieve the same value, that's all that matters.

We see this all the time when using a cell phone. To make a call, the interface is simple—we either dial a number or select an entry in the contact list. Yet, if the provider updates the software, it doesn't change the way you make a call. The interface stays the same regardless of how the implementation changes. However, I can think of one situation when the provider did change the interface—when my area code changed. Fundamental interface changes, like an area code change, do require the users to change behavior. Businesses try to keep these types of changes to a minimum, for some customers will not like the change or perhaps not put up with the hassle.

Recall that in the toaster example, although the interface is always the electric outlet, the implementation could change from a coal power plant to a nuclear power plant without affecting the toaster. One very important caveat should be made here: The coal or nuclear plant must also conform to the interface specification. If the coal plant produces AC power but the nuclear plant produces DC power, a problem exists. The bottom line is that both the user and the implementation must conform to the interface specification.

An Interface/Implementation Example

Let's create a simple (if not very functional) database reader class. We'll write some Java code that will retrieve records from the database. As we've discussed, knowing your end users is always the most important issue when doing any kind of design. You should do some analysis of the situation and conduct interviews with end users, and then list the requirements for the project. The following are some requirements we might want to use for the database reader:

- We must be able to open a connection to the database.
- We must be able to close the connection to the database.
- We must be able to position the cursor on the first record in the database.
- We must be able to position the cursor on the last record in the database.
- We must be able to find the number of records in the database.
- We must be able to determine whether there are more records in the database (that is, if we are at the end).
- We must be able to position the cursor at a specific record by supplying the key.
- We must be able to retrieve a record by supplying a key.
- We must be able to get the next record, based on the position of the cursor.

With these requirements in mind, we can make an initial attempt to design the database reader class by creating possible interfaces for these end users.

In this case, the database reader class is intended for programmers who require use of a database. Thus, the interface is essentially the application-programming interface (API) that the programmer will use. These methods are, in effect, wrappers that enclose the functionality provided by the database system. Why would we do this? We explore this question in much greater detail later in the chapter; the short answer is that we might need to customize some database functionality. For example, we might need to process the objects so that we can write them to a relational database. Writing this *middleware* is not trivial as far as design and coding go, but it is a real-life example of wrapping functionality. More important, we may want to change the database engine itself without having to change the code.

Figure 2.2 shows a class diagram representing a possible interface to the `DataBaseReader` class.

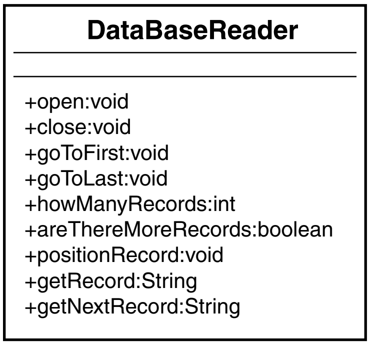

Figure 2.2 A Unified Modeling Language class diagram for the `DataBaseReader` class.

Note that the methods in this class are all public (remember that there are plus signs next to the names of methods that are public interfaces). Also note that only the interface is represented; the implementation is not shown. Take a minute to determine whether this class diagram generally satisfies the requirements outlined earlier for the project. If you find out later that the diagram does not meet all the requirements, that's okay; remember that OO design is an iterative process, so you do not have to get it exactly right the first time.

> **Public Interface**
>
> Remember, an application programmer can access it, and thus, it is considered part of the class interface. Do not confuse the term *interface* with the keyword `interface` used in Java and .NET—which is discussed in later chapters.

For each of the requirements we listed, we need a corresponding method that provides the functionality we want. Now you need to ask a few questions:

- To effectively use this class, do you, as a programmer, need to know anything else about it?
- Do you need to know how the internal database code opens the database?
- Do you need to know how the internal database code physically positions itself over a specific record?
- Do you need to know how the internal database code determines whether any more records are left?

On all counts the answer is a resounding *no!* You don't need to know any of this information. All you care about is that you get the proper return values and that the operations are performed correctly. In fact, the application programmer will most likely be at least one more abstract level away from the implementation. The application will use your classes to open the database, which in turn will invoke the proper database API.

> **Minimal Interface**
>
> Although perhaps extreme, one way to determine the minimalist interface is to initially provide the user no public interfaces. Of course, the class will be useless; however, this forces the user to come back to you and say, "Hey, I need this functionality." Then you can negotiate. Thus, you add interfaces only when it is requested. Never assume that the user needs something.

Creating wrappers might seem like overkill, but there are many advantages to writing them. To illustrate, there are many middleware products on the market today. Consider the problem of mapping objects to a relational database. OO databases have never caught on; however, theoretically they may be perfect for OO applications. However, one small problem exists: Most companies have years of data in legacy relational database systems. How can a company embrace OO technologies and stay on the cutting edge while retaining its data in a relational database?

First, you can convert all your legacy, relational data to a brand-new OO database. However, anyone who has suffered the acute (and chronic) pain of any data conversion knows that this is to be avoided at all costs. Although these conversions can take large amounts of time and effort, all too often they never work properly.

Second, you can use a middleware product to seamlessly map the objects in your application code to a relational model. This is a much better solution since relational databases are so prevalent. Some might argue that OO databases are much more efficient for object persistence than relational databases. In fact, many development systems seamlessly provide this service.

Object Persistence

Object persistence refers to the concept of saving the state of an object so that it can be restored and used at a later time. An object that does not persist basically dies when it goes out of scope. For example, the state of an object can be saved in a database.

However, in the current business environment, relational-to-object mapping is a great solution. Many companies have integrated these technologies. It is common for a company to have a website front-end interface with data on a mainframe.

If you create a totally OO system, an OO database might be a viable (and better performing) option; however, OO databases have not experienced anywhere near the growth that OO languages have.

Standalone Application

Even when creating a new OO application from scratch, it might not be easy to avoid legacy data. Even a newly created OO application is most likely not a standalone application and might need to exchange information stored in relational databases (or any other data storage device, for that matter).

Let's return to the database example. Figure 2.2 shows the public interface to the class, and nothing else. When this class is complete, it will probably contain more methods, and it will certainly contain attributes. However, as a programmer using this class, you do not need to know anything about these private methods and attributes. You certainly don't need to know what the code looks like within the public methods. You simply need to know how to interact with the interfaces.

What would the code for this public interface look like (assume that we start with a Oracle database example)? Let's look at the `open()` method:

```
public void open(String Name){

        /* Some application-specific processing */

        /* call the Oracle API to open the database */

        /* Some more application-specific processing */

};
```

In this case, you, wearing your programmer's hat, realize that the `open` method requires `String` as a parameter. `Name`, which represents a database file, is passed in, but it's not important to explain how `Name` is mapped to a specific database for this example. That's all we need to know. Now comes the fun stuff—what really makes interfaces so great!

Just to annoy our users, let's change the database implementation. Last night we translated all the data from an Oracle database to an SQLAnywhere database (we endured the acute and chronic pain). It took us hours—but we did it.

Now the code looks like this:

```
public void open(String Name){

        /* Some application-specific processing

        /* call the SQLAnywhere API to open the database */

        /* Some more application-specific processing */

};
```

To our great chagrin, this morning not one user complained. This is because even though the implementation changed, the interface did not! As far as the user is concerned, the calls are still the same. The code change for the implementation might have required quite a bit of work (and the module with the one-line code change would have to be rebuilt), but not one line of application code that uses this `DataBaseReader` class needed to change.

Code Recompilation

Dynamically loaded classes are loaded at runtime—not statically linked into an executable file. When using dynamically loaded classes, like Java and .NET do, no user classes would have to be recompiled. However, in statically linked languages such as C++, a link is required to bring in the new class.

By separating the user interface from the implementation, we can save a lot of headaches down the road. In Figure 2.3, the database implementations are transparent to the end users, who see only the interface.

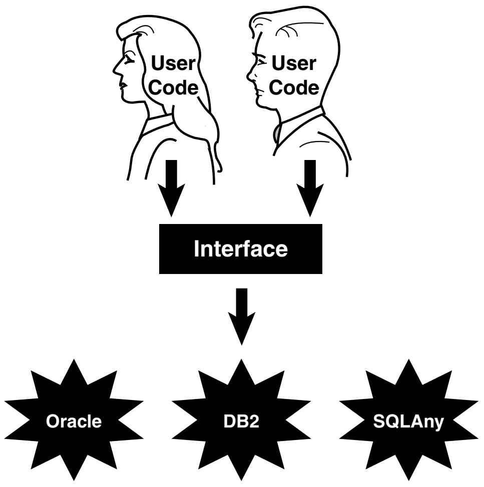

Figure 2.3 The interface.

Using Abstract Thinking When Designing Interfaces

One of the main advantages of OO programming is that classes can be reused. In general, reusable classes tend to have interfaces that are more abstract than concrete. Concrete interfaces tend to be very specific, whereas abstract interfaces are more general. However, simply stating that a highly abstract interface is more useful than a highly concrete interface, although often true, is not always the case.

It is possible to write a very useful, concrete class that is not at all reusable. This happens all the time, and nothing is wrong with it in some situations. However, we are now in the design business and want to take advantage of what OO offers us. So our goal is to design abstract, highly reusable classes—and to do this we will design highly abstract user interfaces. To illustrate the difference between an abstract and a concrete interface, let's create a taxi object. It is much more useful to have an interface such as "drive me to the airport" than to have separate interfaces such as "turn right," "turn left," "start," "stop," and so on, because as illustrated in Figure 2.4, all the user wants to do is get to the airport.

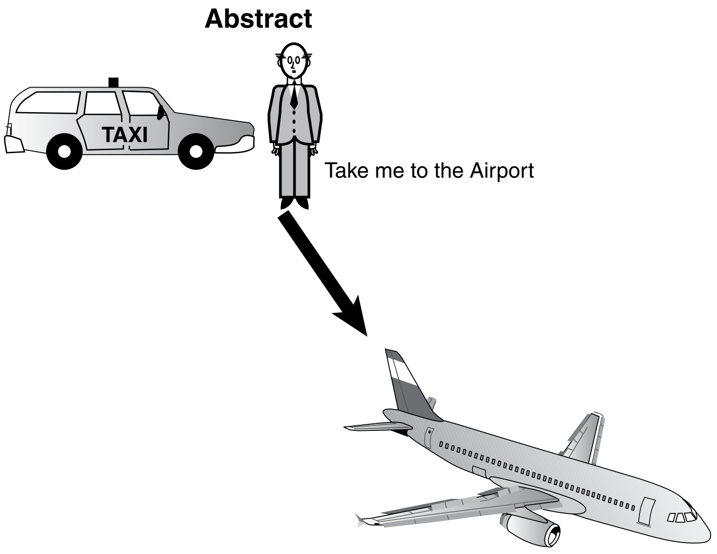

Figure 2.4 An abstract interface.

When you emerge from your hotel, throw your bags into the back seat of the taxi, and get in, the cabbie will turn to you and ask, "Where do you want to go?" You reply, "Please take me to the airport." (This assumes, of course, that there is only one major airport in the city. In Chicago you would have to say, "Please take me to Midway Airport" or "Please take me to O'Hare.") You might not even know how to get to the airport yourself, and even if you did, you wouldn't want to have to tell the cabbie when to turn and which direction to turn, as illustrated in Figure 2.5. How the cabbie implements the actual drive is of no concern to you, the passenger. (However, the fare might become an issue at some point, if the cabbie cheats and takes you the long way to the airport.)

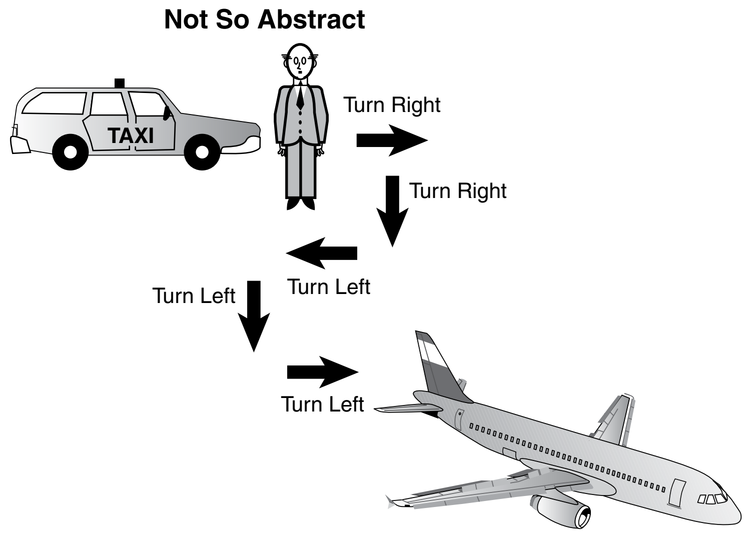

Figure 2.5 A not-so-abstract interface.

Now, where does the connection between abstract and reuse come in? Ask yourself which of these two scenarios is more reusable, the abstract or the not-so-abstract? To put it more simply, which phrase is more reusable: “Take me to the airport,” or “Turn right, then right, then left, then left, then left”? Obviously, the first phrase is more reusable. You can use it in any city, whenever you get into a taxi and want to go to the airport. The second phrase will work only in a specific case. Thus, the abstract interface “Take me to the airport” is generally the way to go for a good, reusable OO design whose implementation would be different in Chicago, New York, or Cleveland.

Providing the Absolute Minimal User Interface Possible

When designing a class, the general rule is to always provide the user with as little knowledge of the inner workings of the class as possible. To accomplish this, follow these simple rules:

- Give the users only what they absolutely need. In effect, this means the class has as few interfaces as possible. When you start designing a class, start with a minimal interface. The design of a class is iterative, so you will soon discover that the minimal set of interfaces might not suffice. This is fine.

- It is better to have to add interfaces because users really need it than to give the users more interfaces than they need. At times it is highly problematic for the user to have access to certain interfaces. For example, you don't want an interface that provides salary information to all users—only the ones who need to know.
- For the moment, let's use a hardware example to illustrate our software example. Imagine handing a user a PC box without a monitor or a keyboard. Obviously, the PC would be of little use. You have just provided the user with the minimal set of interfaces to the PC. However, this minimal set is insufficient, and it immediately becomes necessary to add interfaces.
- Public interfaces define what the users can access. If you initially hide the entire class from the user by making the interfaces private, when programmers start using the class, you will be forced to make certain methods public—these methods thus become the public interface.
- It is vital to design classes from a user's perspective and not from an information systems viewpoint. Too often designers of classes (not to mention any other kind of software) design the class to make it fit into a specific technological model. Even if the designer takes a user's perspective, it is still probably a technician user's perspective, and the class is designed with an eye on getting it to work from a technology standpoint and not from ease of use for the user.
- Make sure when you are designing a class that you go over the requirements and the design with the people who will actually use it—not just developers (this includes all levels of testing). The class will most likely evolve and need to be updated when a prototype of the system is built.

Determining the Users

Let's look again at the taxi example. We have already decided that the users are the ones who will actually use the system. This said, the obvious question is, who are the users?

The first impulse is to say the *customers*. This is only about half right. Although the customers are certainly users, the cabbie must be able to successfully provide the service to the customers. In other words, providing an interface that would, no doubt, please the customer, such as "Take me to the airport for free," is not going to go over well with the cabbie. Thus, in reality, to build a realistic and usable interface, *both* the customer and the cabbie must be considered users.

In short, any object that sends a message to the taxi object is considered a user (and yes, the users are objects, too). Figure 2.6 shows how the cabbie provides a service.

Looking Ahead

The cabbie is most likely an object as well.

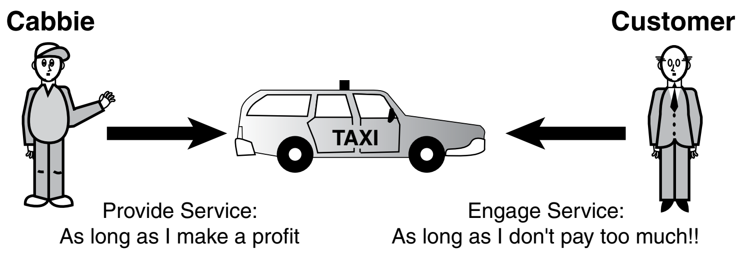

Figure 2.6 Providing services.

Object Behavior

Identifying the users is only a part of the exercise. After the users are identified, you must determine the behaviors of the objects. From the viewpoint of all the users, begin identifying the purpose of each object and what it must do to perform properly. Note that many of the initial choices will not survive the final cut of the public interface. These choices are identified by gathering requirements using various methods such as UML Use Cases.

Environmental Constraints

In their book *Object-Oriented Design in Java*, Gilbert and McCarty point out that the environment often imposes limitations on what an object can do. In fact, environmental constraints are almost always a factor. Computer hardware might limit software functionality. For example, a system might not be connected to a network, or a company might use a specific type of printer. In the taxi example, the cab cannot drive on a road if a bridge is out, even if it provides a quicker way to the airport.

Identifying the Public Interfaces

With all the information gathered about the users, the object behaviors, and the environment, you need to determine the public interfaces for each user object. So think about how you would use the taxi object:

- Get into the taxi.
- Tell the cabbie where you want to go.
- Pay the cabbie.
- Give the cabbie a tip.
- Get out of the taxi.

What do you need to do to use the taxi object?

- Have a place to go.
- Hail a taxi.
- Pay the cabbie money.

Initially, you think about how the object is used and not how it is built. You might discover that the object needs more interfaces, such as "Put luggage in the trunk" or "Enter into a mindless conversation with the cabbie." Figure 2.7 provides a class diagram that lists possible methods for the `Cabbie` class.

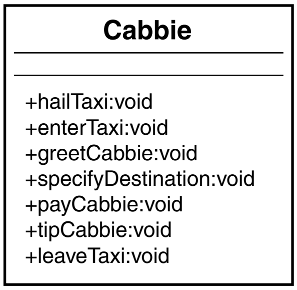

Figure 2.7 The methods in a `Cabbie` class.

As is always the case, nailing down the final interface is an iterative process. For each interface, you must determine whether the interface contributes to the operation of the object. If it does not, perhaps it is not necessary. Many OO texts recommend that each interface model only one behavior. This returns us to the question of how abstract we want to get with the design. If we have an interface called `enterTaxi()`, we certainly do not want `enterTaxi()` to have logic in it to pay the cabbie. If we do this, not only is the design somewhat illogical, but there is virtually no way that a user of the class can tell what has to be done to pay the cabbie.

Identifying the Implementation

After the public interfaces are chosen, you need to identify the implementation. After the class is designed and all the methods required to operate the class properly are in place, the specifics of how to get the class to work are considered.

Technically, anything that is not a public interface can be considered the implementation. This means that the user will never see any of the methods that are considered part of the implementation, including the method's signature (which includes the name of the method and the parameter list), as well as the actual code inside the method.

It is possible to have a private method that is used internally by the class. Any private method is considered part of the implementation given that the user will never see it and thus will not have access to it. For example, a class may have a `changePassword()` method; however, the same class may have a private method that encrypts the password. This method would be hidden from the user and called only from inside the `changePassword()` method.

The implementation is totally hidden from the user. The code within public methods is a part of the implementation because the user cannot see it. (The user should see only the calling structure of an interface—not the code inside it.)

This means that, theoretically, anything that is considered the implementation might change without affecting how the user interfaces with the class. This assumes, of course, that the implementation is providing the answers the user expects.

Whereas the interface represents how the user sees the object, the implementation is really the nuts and bolts of the object. The implementation contains the code that represents that state of an object.

Conclusion

In this chapter, we have explored three areas that can get you started on the path to thinking in an OO way. Remember that there is no firm list of issues pertaining to the OO thought process. Doing things in an OO way is more of an art than a science. Try to think of your own ways to describe OO thinking.

In Chapter 3, "More Object-Oriented Concepts," we discuss the object life cycle: it is born, it lives, and it dies. While it is alive, it might transition through many states. For example, a `DataBaseReader` object is in one state if the database is open and another state if the database is closed. How this is represented depends on the design of the class.

References

Fowler, Martin. 2003. *UML Distilled*, Third Edition. Boston, MA: Addison-Wesley Professional.

Gilbert, Stephen, and Bill McCarty. 1998. *Object-Oriented Design in Java*. Berkeley, CA: The Waite Group Press (Pearson Education).

Meyers, Scott. 2005. *Effective C++*, Third Edition. Boston, MA: Addison-Wesley Professional.

Index

Symbols

{ } (braces), 58

+ (plus sign), 62

/ (slash), comment notations using, 69

A

aborting applications, 54

abstraction
- abstract classes
  - interfaces compared to, 133–135
  - overview of, 121–123, 128–131
- abstract factory design pattern, 165
- abstract interfaces, 41–42
- abstract methods, 129
- nonportable code, 84
- overview of, 25–26, 30–31

accessor methods, 13–14, 73–75

adapter design pattern, 169–171

aggregations
- association and, 153
- concept of, 112–113, 151–152, 153, 180

Alexander, Christopher, 162

Ambler, Scott, 173

analysis, role in system design, 95

antipatterns, 173–174

applications
- aborting, 54
- recovering, 54–55

"The Architecture of Complexity" (Simon), 149

artifacts
- Reuseless, 173
- Robust, 173

associations
- aggregation and, 153
- concept of, 112–113, 152–153
- multiple object, 157–158
- optional, 158–159

attributes
- class, 18, 61–62, 69–71
- importance of, 57–58
- initialization of, 48
- local, 58–59
- object, 12, 59–61
- public versus private, 20
- static, 83

B

base classes, 24–25

behavioral inheritance, 63

behavioral patterns
- categories of, 171–172
- iterator, 172–173

behaviors, object. ***See also*** **methods**
- overview of, 13–16, 44
- separating out, 200–202

bitwise copies, 64

builder design pattern, 165

buyInventory() method, 142

C

C / C++ development. ***See*** **OO (object-oriented) development**

C# development. ***See*** **OO (object-oriented) development**

C++ Report, **173**

Cabbie class
- accessors, 73
- attributes, 69–71
- class diagram, 115–116
- comments, 69
- constructors
  - default, 49
  - example of, 47–48
  - overview of, 71–72
- name, 68
- overloaded methods, 50

calcArea() method, 188–189

CalculateAreas class, 189, 190–191, 192

CalculatePay() method, 14

calling constructors, 48

Car class, 137, 138

cardinality, 155–157

Cat class, 199, 202

catch keyword, 55–57

catching exceptions, 56–57

categories, design pattern, 164–165

CatNoise class, 201, 204

chain of response, 171

CIRCLE class, 167–169

Circle class, 27–30, 119, 130–131, 189

class diagrams
- cardinality in, 156–157
- composition, 113–114
- creating, 51–52
- DataBaseReader class, 51
- Dog class, 159
- e-business case study, 142–146
  - CustList class, 143
  - DonutShop class, 143–144
  - PizzaShop class, 144
  - Shop class, 142
- overview of, 19–20, 92

class keyword, 67–69
classes. *See also* interfaces; methods; objects
abstract. *See also* abstraction
interfaces compared to, 133–135
overview of, 121–123, 128–131
attributes
class scope, 61–62
example of, 69–71
initialization of, 48
local scope, 58–59
object scope, 59–61
overview of, 18, 61–62
public versus private, 20
Cabbie
accessors, 73
attributes, 69–71
class diagram, 115–116
comments, 69
constructors, 47–49, 71–72
name, 68
overloaded methods, 50
CalculateAreas, 189, 190–191, 192
Car, 137, 138
Cat, 199, 202
CatNoise, 201, 204
Circle, 27–30, 119, 130–131, 167–169, 189
comments
design guidelines, 81–82
notation, 69
number of, 82
composition
advantages of, 175–177
aggregations, 112–113, 151–152, 153
associations, 112–113, 152–153, 157–159
building in phases, 149–151
cardinality, 155–157
class diagrams, 113–114
definition of, 105
dependencies, 154–155
example of, 112, 159–160
object reuse, 105–106
overview of, 30
relationships, 148–149
constructors
calling, 48
default, 48–49
design of, 53–54, 80–81
example of, 71–72
injection by, 184
lack of, 71
multiple, 49–50, 72
overview of, 47–48
purpose of, 48
return values, 47
Count, 49–50
CustList, 143
data hiding, 20
database reader example, 36–40
DataBaseReader
class diagrams, 51
constructors, 52
overview of, 36–40
testing, 87–88
definition of, 16–17
design guidelines, 10
code reuse, 82
comments, 81–82
constructors/destructors, 80–81
copying and comparison, 84
extensibility, 83
implementation, hiding, 79–80
interaction, 82
maintainability, 86
marshalling, 89
naming conventions, 83–84

nonportable code, 84
object persistence, 88–89
public interface, 78–79
real-world system modeling, 77–78
scope, 84–85
serialization, 89
stubs, 86–88
top-down, 77
destructors, 80–81
Dog
class diagram, 159–160
contract, 137–138
defining, 134
dependency injection, 182–184
design decisions, 110–112
generalization-specialization, 109–110
inheritance, 107–109
DonutShop, 143–144
Employee
behaviors, 13–16
cardinality, 155–157
multiple object associations, 157–158
optional associations, 158–159
encapsulation
importance of, 113–114
inheritance weakened by, 115–117
overview of, 20
error handling
applications, aborting, 54
design guidelines, 81
exceptions, catching, 56–57
exceptions, throwing, 55–57
ignoring problems, 54
overview of, 54
recovery, 54–55
Head, 133
highly coupled, 86, 175–177. *See also* dependencies
identifying, 96
implementation
characteristics of, 36
hiding, 79–80
inheritance
advantages and limitations, 106–109
behavioral, 63
composition as alternative to, 175–177, 179–182
definition of, 105
design decisions, 110–112
example of, 159–160, 177–179
generalization-specialization, 109–110
implementation, 63
multiple, 26, 63, 131–132
object reuse, 105–106
overview of, 23–24
polymorphism, 27–30
relationships, 131, 147
single, 26
subclasses, 24–25
superclasses, 24–25
weakened by encapsulation, 115–117
Integer, 169–170
interface/implementation paradigm
model of, 22–23
overview of, 21
real-world example, 21–22
interfaces
abstract, 41–42
characteristics of, 36
database reader example, 36–40
design guidelines, 41–42
extending, 79
IMammal, 197

implementation versus, 34–35
ISP (Interface Segregation Principle), 197–198
IWalkable, 183–184
minimum public, 78–79
overview of, 20–21, 131–132
prototypes, 97
public, 44–45, 75
testing, 86–88
Iterator, 172
MailTool, 170
MainApplication, 98
MakingNoise, 200
Mammal, 203
composition, 179–182
defining, 133
inheritance, 178–179
interfaces for, 197–198
makeNoise() method, 199
Math, 85
messages, 19
model of, 96–97
MyMailTool, 170–171
names, 67–69
Number
class attributes, 61–62
local attributes, 58–59
object attributes, 59–61
objects, creating from, 17–18
OpenClosed, 192, 194
Person
attributes, 18
class diagram, 19–20
creating, 18
extensibility, 83
methods, 19
PizzaShop, 144
Planet, 136, 137
polymorphism
object responsibility, 118–119
overview of, 117
Rectangle, 117, 119, 130–131, 192, 194–197
references, 64
relationships
has-a, 31
is-a, 26–27, 107
scope, 84–85
setters, injection by, 184
Shape, 165
calcArea() method, 188
child classes, 167–168
class hierarchy, 128–131
factory method design pattern, 165–169
generate() method, 167
is-a relationships, 26–27
polymorphism, 27–30, 117–121
ShapeFactory class, 168–169
ShapeCalculator, 192
ShapeFactory, 168–169
Shop, 142
SomeMath, 100
Sound, 101
Square, 22–23, 167–169
Star, 119
superclasses, 53
Swimmable, 181
TestBeep, 101
TestFactoryPattern, 169
TestMammal, 200, 201–202, 203–204
TestMath, 100
TestShape, 119–121, 190, 191
TestShop, 145–146
Triangle, 120, 167–169
Whale, 181

Window, 117
wrapping, 101–102
Coad, Peter, 106, 115
code reuse. *See also* abstraction; object reuse
advantages and limitations, 125
contracts
defining, 136–138
overview of, 128
as system plug-in points, 138–139
design guidelines for, 82
e-business case study
code reuse for, 141–142
non-reuse approach, 139–141
scenario, 139
UML object model, 142–146
frameworks, 126–127
interfaces
abstract, 41–42
abstract classes compared to, 133–135
characteristics of, 36
database reader example, 36–40
design guidelines, 41–42
extending, 79
IMammal, 197
implementation versus, 34–35
interface/implementation paradigm, 21–23
is-a relationships, 135–136
ISP (Interface Segregation Principle), 197–198
IWalkable, 183–184
minimum public interface, 78–79
overview of, 20–21, 131–132
prototypes, 97
public, 44–45, 75
terminology, 131
testing, 86–88
UML diagrams, 132
command design pattern, 171
comments
design guidelines, 81–82
notation, 69
number of, 82
communication, object-to-object, 10–11
comparing objects, 84
composition, 30
advantages of, 175–177
aggregations
association and, 153
concept of, 112–113, 151–152, 153
associations
aggregation and, 153
concept of, 112–113, 152–153
multiple object, 157–158
optional, 158–159
building in phases, 149–151
cardinality, 155–157
class diagrams, 113–114
definition of, 105
dependencies, avoiding, 154–155
example of, 112, 159–160, 179–182
object reuse, 105–106
relationships, 148–149
concatenation of strings, 62
conditions, 98–99
consequences, 162
constraints, environmental, 44
constructors
calling, 48
default, 48–49
design of, 53–54, 80–81
example of, 71–72
injection, 80

injection by, 184, 199
lack of, 71
multiple, 49–50, 72
overview of, 47–48
purpose of, 48
return values, 47

contracts
defining, 136–138
overview of, 128
as system plug-in points, 138–139

copies, 64–65

copying objects, 84

Count class, 49–50

coupling, 86, 175–177

"Creating Chaos" (Johnson), 173

creational patterns
categories of, 165
factory method, 165

curly braces ({}), 58

CustList class, 143

customers, 79

D

data hiding, 9, 20

data transfer objects (DTOs), 78

DataBaseReader class
class diagram, 51
constructors, 52
overview of, 36–40
testing, 87–88

databases. *See also* DataBaseReader class
NoSQL, 89
relational, 89

declaring methods
private implementation methods, 76
public interface methods, 75
static, 73–74

decoupling. *See* dependencies

deep copies, 64

default constructors, 48–49

definition inheritance. *See* inheritance

dependencies, 154–155. *See also* dependency injection; inheritance
composition, 30
advantages of, 175–177
example of, 179–182
DIP (Dependency Inversion Principle), 3
dependency injection and, 202–204
initial example, 199–200
overview of, 198–199
separating out behavior, 200–202
inheritance
composition as alternative to, 175–177, 179–182
issues with, 179–182

dependency injection, 182–184
by constructor, 80, 184, 199
definition of, 198
DIP (Dependency Inversion Principle), 3, 202–204
initial example, 199–200
overview of, 198–199
separating out behavior, 200–202
example of, 182–184
by parameters, 199
by setter, 184

Dependency Inversion Principle. *See* DIP (Dependency Inversion Principle)

design
classes, 10
code reuse, 82
comments, 81–82
constructors/destructors, 80–81
copying and comparison, 84
extensibility, 83

identifying, 96
implementation, hiding, 79–80
interaction, 82
maintainability, 86
naming conventions, 83–84
nonportable code, 84
public interface, 78–79
real-world system modeling, 77–78
scope, 84–85
comments, 81–82
constructors, 53–54
error handling, 81
global data, 8–9, 85
guidelines and best practices
iteration in, 86
marshalling, 89
object persistence, 88–89
serialization, 89
stubs, 86–88
top-down design, 77
inheritance, 110–112
interfaces, 41–42
objects, 12
patterns
adapter, 169–171
advantages of, 162
antipatterns, 173–174
best practices, 161
categories of, 164–165
elements of, 162
factory method, 165–169
iterator, 172–173
MVC (Model/View/Controller), 163–164
overview of, 161–162
SOLID principles
DIP (Dependency Inversion Principle), 198–204
ISP (Interface Segregation Principle), 197–198
LSP (Liskov Substitution Principle), 194–197
OCP (Open/Close Principle), 192–194
overview of, 187–188
SRP (Single Responsibility Principle), 187–188
system
analysis, 95
building in phases, 149–151
class identification, 96
class model, 96–97
object wrappers, 97–102
OO design process, 91–94
requirements documents, 95
safety versus economics, 94
SOW (statement of work), 95
system prototypes, 96
user interface prototypes, 97
waterfall model, 92–93
design patterns
adapter, 169–171
advantages of, 162
antipatterns, 173–174
best practices, 161
categories of, 164–165
elements of, 162
factory method, 165–169
iterator, 172–173
MVC (Model/View/Controller), 163–164
overview of, 161–162
Design Patterns **(Gamma et al), 161–162. *See also* design patterns**
destructors, 80–81
diagrams, class
cardinality in, 156–157
composition, 113–114

creating, 51–52
DataBaseReader class, 37, 51
Dog class, 156–157, 159
e-business case study, 142–146
CustList class, 143
DonutShop class, 143–144
PizzaShop class, 144
Shop class, 142
overview of, 19–20, 92
diagrams, interface, 132
Dictionary.com, 128
DIP (Dependency Inversion Principle), 3
dependency injection, 202–204
initial example, 199–200
overview of, 198–199
separating out behavior, 200–202
documentation. *See also* diagrams, class
amount of, 82
comments
design guidelines, 81–82
notation, 69
requirements documents, 95
SOW (statement of work), 95
Dog class, 182, 184
class diagram, 159–160
defining, 134, 137, 138
design decisions, 110–112
generalization-specialization, 109–110
inheritance, 107–109
domains, mixing, 155
DonutShop class, 143–144
draw() method, 119, 128–131
DTOs (data transfer objects), 78

E

e-business case study
code reuse for, 141–142
non-reuse approach, 139–141
scenario, 139
UML object model, 142–146
CustList class, 143
DonutShop class, 143–144
PizzaShop class, 144
Shop class, 142
economics, safety versus, 94
***Effective C++* (Meyers), 63, 78, 109**
Employee class
behaviors, 13–16
cardinality, 155–157
multiple object associations, 157–158
optional associations, 158–159
Employee object, 14
encapsulation
definition of, 10
importance of, 113–114
inheritance weakened by, 115–117
overview of, 20
enums, 167
environmental constraints, 44
error handling
aborting application, 54
design guidelines, 81
exceptions
catching, 56–57
throwing, 55–57
ignoring problems, 54
overview of, 54
recovery, 54–55
exceptions
catching, 56–57
throwing, 55–57
extensibility
design guidelines, 83
interfaces, 79

F

factory method design pattern, 165–169

flat file systems, 89

fragility, 187

frameworks, 126–127

G

Gamma, Erich, 161–162

Gang of Four, 161–162

garbage collection, 80

generalization-specialization, 109–110

generate() method, 167–168

generateHeat() method, 133

getArea() method, 29–30

getInventory() method, 142

getMail() method, 171

getSize() method, 133

getSocialSecurityNumber(), 14

getters, 13–14, 73–74

Gilbert, Stephen, 115

giveDestination() method, 75, 76

global data, 8–9, 85

GoF (Gang of Four), 161–162

H

handling errors. *See* error handling

has-a relationships, 31

hasMoreElements() method, 173

Head class, 133

Helm, Richard, 161–162

hiding

  data, 9

  implementation, 79–80

highly coupled classes, 86, 175–177. *See also* dependencies

hybrid apps, 7

I

ignoring problems, 54

IMammal interface, 197

immobility, 187

implementations. *See also* inheritance

  characteristics of, 36

  database reader example, 36–40

  hiding, 79–80

  identifying, 45–46

  interface/implementation paradigm

    model of, 22–23

    overview of, 21

    real-world example, 21–22

  interfaces versus, 34–35

  private implementation methods, 76

inheritance. *See also* composition; encapsulation

  advantages and limitations, 106–109

  behavioral, 63

  composition as alternative to, 175–177, 179–182

  definition of, 105

  design decisions, 110–112

  example of, 159–160

  generalization-specialization, 109–110

  implementation, 63

  is-a relationships, 107

  multiple, 26, 63, 131–132

  object reuse, 105–106

  overview of, 23–24

  polymorphism, 27–30

  relationships, 26–27, 131, 135–136, 147

  single, 26

  weakened by encapsulation, 115–117

init keyword, 47

initialization, attribute, 48

injection, dependency. *See* dependency injection

Integer class, 169–170

interaction, design guidelines for, 82

Interface Segregation Principle (ISP), 3, 197–198

interfaces

- abstract, 41–42
- abstract classes compared to, 133–135
- characteristics of, 36
- database reader example, 36–40
- design guidelines, 41–42
- extending, 79
- IMammal, 197
- interface/implementation paradigm, 34–35
  - model of, 22–23
  - overview of, 21
  - real-world example, 21–22
- is-a relationships, 135–136
- ISP (Interface Segregation Principle), 197–198
- IWalkable, 183–184
- Nameable, 132, 136, 137
- overview of, 20–21, 131–132
- prototypes, 97
- public, 44–45
  - methods, 75
  - minimum public interface, 78–79
- terminology, 131
- testing, 86–88
- UML diagrams, 132

internal customers, 79

interpreter design pattern, 171

Inversion of Control (IoC), 72

IOC (inversion of control), 182

IPS. *See* Interface Segregation Principle (ISP)

is-a relationships, 26–27, 107, 135–136

ISP (Interface Segregation Principle), 3, 197–198

iterate() method, 173

iterations, 86, 99

Iterator class, 172

iterator design pattern, 172–173

IWalkable interface, 183–184

J

Java. *See* OO (object-oriented) development

***Java Design* (Coad and Mayfield), 106**

Java development. *See* OO (object-oriented) development

***Java Primer Plus* (Tyma, Torok, and Downing), 54**

Johnson, Johnny, 173

Johnson, Ralph, 161–162

K

keywords. *See also* methods

- catch, 55–57
- class, 68
- classes, 67–69
- init, 47
- new, 47, 53, 165, 169, 181
- null, 71–72
- private, 69–71, 76
- public, 75–76
- static, 61–62, 69–71, 74–75
- this, 60
- try, 55–57

Koenig, Andrew, 173

L

Larman, Craig, 1

leaks, memory, 81

legacy systems, OO (object-oriented) concepts with, 6–7

Liskov Substitution Principle (LSP), 3, 109, 194–197

local attributes, 58–59
LSP. *See* Liskov Substitution Principle (LSP)

M

MailTool class, 170
MainApplication class, 98
maintainability, 86
makeNoise() method, 133, 199
MakingNoise class, 200
Mammal class, 203
  composition, 179–182
  defining, 133
  inheritance, 178–179
  interfaces for, 197–198
  makeNoise() method, 199
marshalling objects, 89
Martin, Robert, 7, 187
Math class, 85
Mayfield, Mark, 106, 115
McMarty, Bill, 115
mediator design pattern, 172
memento design pattern, 172
memory leaks, 81
messages, 19
methods, 13. *See also* keywords
  abstract, 129
  accessors, 13–14, 73–75
  buyInventory(), 142
  calcArea(), 188–189
  CalculatePay(), 14
  constructors
    calling, 48
    default, 48–49
    design of, 53–54, 80–81
    example of, 71–72
    injection by, 184
    lack of, 71
    multiple, 49–50, 72
    overview of, 47–48
    purpose of, 48
    return values, 47
  destructors, 80–81
  draw(), 119, 128–131
  generate(), 167–168
  generateHeat(), 133
  getArea(), 29–30
  getInventory(), 142
  getMail(), 171
  getSize(), 133
  getSocialSecurityNumber(), 14
  getters, 13–14, 73–74
  giveDestination(), 75, 76
  hasMoreElements(), 173
  iterate(), 173
  makeNoise(), 133, 199
  mutators, 13–14
  open(), 39–40
  overloading, 50–51
  overview of, 19
  private implementation, 76
  public interface, 75
  retrieveMail(), 170
  setSize(), 133
  setters, 13–14, 73–74, 184
  setWalker(), 184–185
  signatures, 50–51
  static, 74–75, 83
  turnRight(), 76
  virtual, 121–123
  walk(), 183
Meyers, Scott, 63, 78, 109
middleware, 37–39
minimum public interface, 78–79
mobile web, 7
modeling tools, 15

Model/View/Controller (MVC) design pattern, 163–164
multiple constructors, 49–50, 72
multiple inheritance, 26, 63, 131–132
multiple object associations, 157–158
mutator methods, 13–14
MVC (Model/View/Controller) design pattern, 163–164
MyMailTool class, 170–171

N

Nameable interface, 132, 136, 137
naming conventions
  classes, 67–69
  design guidelines for, 83–84
  patterns, 162
new keyword, 47, 53, 165, 169, 181
nonportable code, 84, 101
NoSQL databases, 89
null value, 71–72
Number class
  class attributes, 61–62
  local attributes, 58–59
  object attributes, 59–61

O

object attributes, 59–61
***The Object Primer* (Ambler), 86**
object reuse, 105–106, 204
  composition, 30
    advantages of, 175–177
    aggregation, 112–113, 151–152
    aggregations, 153
    association, 112–113
    associations, 152–153, 157–159
    building in phases, 149–151
    cardinality, 155–157
    class diagrams, 113–114
    definition of, 105
    dependencies, avoiding, 154–155
    example of, 112, 159–160, 179–182
    object reuse, 105–106
    relationships, 148–149
  inheritance. *See also* composition
    behavioral, 63
    composition as alternative to, 175–177, 179–182
    definition of, 105
    design decisions, 110–112
    example of, 159–160
    generalization-specialization, 109–110
    implementation, 63
    is-a relationships, 26–27
    multiple, 26, 63
    object reuse, 105–106
    overview of, 23–24
    polymorphism, 27–30
    relationships, 147
    single, 26
object wrappers
  definition of, 7
  design guidelines, 97–98
  for existing classes, 101–102
  for nonportable code, 101
  overview of, 97–98
  for structured code, 98–100
Objective-C, 2
***Object-Oriented Design in Java* (Gilbert and McCarty), 44, 54, 64, 78, 155**
object-oriented development. *See* OO (object-oriented) development
objects. *See also* classes; methods; object reuse
  attributes, 12
    class scope, 61–62
    example of, 69–71

initialization of, 48
local, 58–59
object scope, 59–61
public versus private, 20
behaviors, 13–16, 44
comparing, 84
copies, 64–65
copying, 84
creating, 17–18
definition of, 8, 12
design, 12
Employee, 14
marshalling, 89
object-to-object communication, 10–11
operations, 63–65
Payroll, 14
persistence, 39, 88–89
properties, 13
responsibility, 118–119
scope
class attributes, 61–62
importance of, 57–58
local attributes, 58–59
object attributes, 59–61
serialization, 89
wrappers
definition of, 7
design guidelines, 97–98

observer design pattern, 172

OCP. *See* Open/Close Principle

OO (object-oriented) development, 11. *See also* abstraction; classes; code reuse; dependencies; objects
abstraction
abstract classes, 121–123, 128–131, 133–135
abstract factory design pattern, 165
abstract interfaces, 41–42
abstract methods, 129
nonportable code, 84
overview of, 25–26, 30–31
advantages of, 11–12
comments
design guidelines, 81–82
notation, 69
number of, 82
composition, 30
advantages of, 175–177
aggregations, 112–113, 151–152, 153
associations, 112–113, 152–153, 157–159
building in phases, 149–151
cardinality, 155–157
class diagrams, 113–114
definition of, 105
dependencies, avoiding, 154–155
example of, 112, 159–160, 179–182
object reuse, 105–106
relationships, 148–149
contracts
defining, 136–138
overview of, 128
as system plug-in points, 138–139
data hiding, 9
e-business case study
code reuse for, 141–142
non-reuse approach, 139–141
scenario, 139
UML object model, 142–146
encapsulation
definition of, 10
importance of, 113–114
inheritance weakened by, 115–117
environmental constraints, 44
error handling
aborting application, 54
design guidelines, 81

exceptions, catching, 56–57
exceptions, throwing, 55–57
ignoring problems, 54
overview of, 54
recovery, 54–55
evolution of, 5
frameworks, 126–127
implementations
characteristics of, 36
database reader example, 36–40
hiding, 79–80
identifying, 45–46
interface/implementation paradigm, 21–23
interfaces versus, 34–35
private implementation methods, 76
inheritance
advantages and limitations, 106–109
behavioral, 63
composition as alternative to, 175–177, 179–182
definition of, 105
design decisions, 110–112
example of, 159–160, 177–179
generalization-specialization, 109–110
implementation, 63
multiple, 26, 63, 131–132
object reuse, 105–106
overview of, 23–24
polymorphism, 27–30
relationships, 131, 135–136, 147
single, 26
subclasses, 24–25
superclasses, 24–25
weakened by encapsulation, 115–117
interface/implementation paradigm
model of, 22–23
overview of, 21
real-world example, 21–22
interfaces
abstract, 41–42
abstract classes compared to, 133–135
characteristics of, 36
database reader example, 36–40
design guidelines, 41–42
extending, 79
IMammal, 197
implementation versus, 34–35
interface/implementation paradigm, 21–23
is-a relationships, 135–136
ISP (Interface Segregation Principle), 197–198
IWalkable, 183–184
minimum public interface, 78–79
overview of, 20–21, 131–132
prototypes, 97
public, 44–45, 75
terminology, 131
testing, 86–88
UML diagrams, 132
iteration in, 86
legacy systems and, 6–7
object-to-object communication, 10–11
operators, overloading, 62–63
polymorphism
object responsibility, 118–119
overview of, 117
procedural programming compared to, 7–11
relationships
has-a, 31
is-a, 26–27, 107

scope
class attributes, 61–62
design guidelines, 84–85
importance of, 57–58
local attributes, 58–59
object attributes, 59–61
SOLID principles
DIP (Dependency Inversion Principle), 198–204
ISP (Interface Segregation Principle), 197–198
LSP (Liskov Substitution Principle), 194–197
OCP (Open/Close Principle), 192–194
overview of, 187–188
SRP (Single Responsibility Principle), 188–191
stacks, 29
users, determining, 43–44
open() method, 39–40
Open/Close Principle, 192–194
Open/Close Principle (OCP), 3, 192–194
OpenClosed class, 192, 194
operations, object, 63–65
operators, overloading, 62–63
optional associations, 158–159
overloading
methods, 50–51
operators, 62–63

P

parameters, injection by, 199
parent class, 24–25
passing references, 71
***A Pattern Language* (Alexander), 162**
patterns, design
adapter, 169–171
advantages of, 162
antipatterns, 173–174
best practices, 161
categories of, 164–165
elements of, 162
factory method, 165–169
iterator, 172–173
MVC (Model/View/Controller), 163–164
overview of, 161–162
Payroll object, 14
persistence, 39, 88–89
Person class
attributes, 18
class diagram, 19–20
creating, 18
extensibility, 83
methods, 19
PizzaShop class, 144
Planet class, 136, 137
plus sign (+), 62
polymorphism
object responsibility, 118–119
overview of, 27–30, 117
private attributes, 20
private implementation methods, 76
private keyword, 69–71, 76
problems, 162
procedural programming
data model, 11
OO (object-oriented) programming compared to, 7–11
properties, object, 13
protocols, 121–123
prototype design pattern, 165
prototypes
system, 96
user interface, 97–98
public attributes, 20
public interfaces, 44–45, 75
public keyword, 75–76

Q-R

recovery, 54–55
Rectangle class, 117, 119, 130–131, 192, 194–197
references
  classes and, 64
  passing, 71
relational databases, 89
relationships
  composition, 148–149
  has-a, 31
  inheritance, 131, 147
  is-a, 26–27, 107, 135–136
requirements documents, 95
responsibility, SRP (Single Responsibility Principle), 187–188
retrieveMail() method, 170
return values, 47
reuse of code. *See* code reuse
“Reuse Patterns and Antipatterns” (Ambler), 173
Reuseless Artifact, 173
rigidity, 187
Robust Artifacts, 173

S

safety, economics versus, 94
scope
  class attributes, 61–62
  design guidelines, 84–85
  importance of, 57–58
  local attributes, 58–59
  object attributes, 59–61
separating out behavior, 200–202
sequences, 98–99
serialization, 89
setSize() method, 133
setters, 13–14, 73–74, 184
setWalker() method, 184–185
shallow copies, 64
Shape class, 165
  calcArea() method, 188
  child classes, 167–168
  class hierarchy, 128–131
  factory method design pattern, 165–169
  generate() method, 167
  is-a relationships, 26–27
  polymorphism, 27–30, 117–121
  ShapeFactory class, 168–169
ShapeCalculator class, 192
ShapeFactory class, 168–169
ShapeType enum, 167
Shop class, 142
signatures, 21, 50–51
Simon, Herbert, 149
single inheritance, 26
Single Responsibility Principle (SRP), 3, 187–188
singleton design pattern, 165
slash (/), 69
Smalltalk
  development of, 163
  MVC (Model/View/Controller) design pattern, 164–165
SOLID principles, 2–3, 109
  DIP (Dependency Inversion Principle)
    dependency injection, 202–204
    initial example, 199–200
    overview of, 198–199
    separating out behavior, 200–202
  ISP (Interface Segregation Principle), 197–198
  LSP (Liskov Substitution Principle), 194–197
  OCP (Open/Close Principle), 192–194

- overview of, 187–188
- SRP (Single Responsibility Principle), 187–188

solutions, 162

SomeMath class, 100

Sound class, 101

SOW (statement of work), 95

specialization, 109–110

Square class, 22–23, 167–169

SRP. *See* Single Responsibility Principle (SRP)

stacks, 29

standalone applications, 39

Star class, 119

state design pattern, 172

statement of work (SOW), 95

static attributes, 83

static keyword, 61–62, 69–71, 74–75

static methods, 83

strategy design pattern, 172

strings, concatenation of, 62

structural patterns
- adapter, 169–171
- categories of, 169

structured code
- conditions, 98–99
- sequences, 98–99
- wrapping, 99–100

stubs, 86–88

subclasses, 24–25

substitution, LSP (Liskov Substitution Principle), 194–197

superclasses, 24–25, 53

Swift
- exceptions, 55–57
- init keyword, 47
- multiple inheritance, 63
- scope, 58

Swimmable class, 181

system design
- analysis, 95
- building in phases, 149–151
- class identification, 96
- class model, 96–97
- object wrappers, 97–98
  - for existing classes, 101–102
  - for nonportable code, 101
  - overview of, 97–98
  - for structured code, 98–100
- OO design process, 91–94
- requirements documents, 95
- safety versus economics, 94
- SOW (statement of work), 95
- system prototypes, 96
- user interface prototypes, 97
- waterfall model, 92–93

system prototypes, 96

T

template method, 172

TestBeep class, 101

TestFactoryPattern class, 169

testing interfaces, 86–88

TestMammal class, 200, 201–202, 203–204

TestMath class, 100

TestShape class, 119–121, 190, 191

TestShop class, 145–146

this keyword, 60

throwing exceptions, 55–57

top-down design, 77

Triangle class, 120, 167–169

troubleshooting. *See* error handling

try keyword, 55–57

try/catch blocks, 55–57

turnRight() method, 76

U

UML (Unified Modeling Language)
- class diagrams, 14–15, 19–20, 92
  - creating, 51–52
  - DataBaseReader, 37
- interface diagrams, 132

user interface prototypes, 97

users
- customers versus, 79
- determining, 43–44

V

variables, global, 85

virtual methods, 121–123

visitor design pattern, 172

Visual Basic .NET, 2
- exceptions, 55–57
- multiple inheritance, 63
- New keyword, 47
- operator overloading, 63

Vlissides, John, 161–162

W-X-Y-Z

walk() method, 183

waterfall model, 92–93

Whale class, 181

Window class, 117

word processing framework, 126–127

wrappers
- advantages of, 38
- design guidelines, 97–98
- for existing classes, 101–102
- for nonportable code, 101
- overview of, 7, 97–98
- for structured code, 98–100

Xerox PARC, 163