Matt Weisfeld

Fifth Edition

The Object-Oriented Thought Process

About This E-Book

EPUB is an open, industry-standard format for e-books. However, support for EPUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font, font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer's Web site.

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the e-book in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a "Click here to view code image" link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

The Object-Oriented Thought Process

Fifth Edition

The Pearson Addison-Wesley Developer's Library

Visit **informit.com/devlibrary** for a complete list of available publications.

The **Developer's Library** series from Pearson Addison-Wesley provides practicing programmers with unique, high-quality references and tutorials on the latest programming languages and technologies they use in their daily work. All books in the Developer's Library are written by expert technology practioners who are exceptionally skilled at organizing and presenting information in a way that is useful for other programmers.

Developer's Library titles cover a wide range of topics, from open source programming languages and technologies, mobile application

source programming languages and technologies, mobile application developments, and web development to Java programming and more.

Make sure to connect with us!
informit.com/socialconnect

Pearson
Addison-Wesley

informIT.com
the trusted technology learning source

O'REILLY
Safari

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

Library of Congress Control Number: 2019930825

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

ISBN-10: 0-13-518196-8

3 20

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

Table of Contents

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.

Introduction

This Book's Scope

As the title suggests, this book is about the object-oriented (OO) thought process. Although choosing the theme and title of a book are important decisions, these decisions are not at all straightforward when dealing with a highly conceptual topic. Many books deal with one level or another of programming and object orientation. Several popular books cover topics including OO analysis, OO design, OO programming, design patterns, OO data (XML), the Unified Modeling Language (UML), OO web development, OO mobile development, various OO programming languages, and many other topics related to OO programming (OOP).

However, while poring over all these books, many people forget that all these topics are built on a single foundation: how you think in OO ways. Often, many software professionals, as well as students, dive into these books without taking the appropriate time and effort to really understand the design concepts behind the code.

I contend that learning OO concepts is not accomplished by learning a specific development method, a programming language, or a set of design tools. Object-oriented development is, simply put, a way of thinking. This book is all about the OO thought process.

Separating the languages, development practices, and tools from the OO thought process is not an easy task. Often, people are introduced to OO concepts by diving headfirst into a programming language. For example, many years ago a large number of C programmers were first introduced to object orientation by migrating directly to C++ before they were even remotely exposed to OO concepts.

It is important to understand the significant difference between learning OO concepts and programming in an OO language. This came into sharp focus for

me well before I worked on the first edition of this book, when I read articles like Craig Larman's "What the UML Is—and Isn't," In this article he states,

> Unfortunately, in the context of software engineering and the UML diagramming language, acquiring the skills to read and write UML notation seems to sometimes be equated with skill in object-oriented analysis and design. Of course, this is not so, and the latter is much more important than the former. Therefore, I recommend seeking education and educational materials in which intellectual skill in object-oriented analysis and design is paramount rather than UML notation or the use of a case tool.

Thus, although learning a modeling language is an important step, it is much more important to learn OO skills first. Learning UML before fully understanding OO concepts is similar to learning how to read an electrical diagram without first knowing anything about electricity.

The same problem occurs with programming languages. As stated earlier, many C programmers moved into the realm of object orientation by migrating to C++ before being directly exposed to OO concepts. This would always come out in an interview. Quite often developers who claim to be C++ programmers are simply C programmers using C++ compilers. Even now, with languages such as C# .NET, VB .NET, Objective-C, Swift, and Java well established, a few key questions in a job interview can quickly uncover a lack of OO understanding.

Early versions of Visual Basic are not OO. C is not OO, and C++ was developed to be backward compatible with C. Because of this, it is quite possible to use a C++ compiler while using only C syntax while forsaking all of C++'s OO features. Objective-C was designed as an extension to the standard ANSI C language. Even worse, a programmer can use just enough OO features to make a program incomprehensible to OO and non-OO programmers alike.

Thus, it is of vital importance that while you're learning to use OO development environments, you first learn the fundamental OO concepts. Resist the temptation to jump directly into a programming language, and instead take the time to learn the object-oriented thought process first.

What's New in the Fifth Edition

As stated often in this introduction, my vision for the first edition was to stick to the concepts rather than focus on a specific emerging technology. Although I still adhere to this goal for the fifth edition, I also introduce more of the "counter-arguments" than were present in the earlier editions. By that I mean that although object-oriented development is, by far, the biggest game in town, it is not the only one.

Since the first edition of this book was completed in 1999, many technologies have emerged and some have faded. At the time, Java was just establishing itself and was the primary OO development language. Web pages would soon become a part of daily life and business. We all know how ubiquitous mobile devices have become. In the past 20 years software developers have encountered XML, JSON, CSS, XSLT, SOAP, and RESTful Web Services. Android devices use Java and now Kotlin, while iOS devices use Objective-C and Swift.

The point I am trying to make is that we have embraced a lot of technologies in the past 20 years (and four editions of the book). My primary goal for this edition is to condense all of this down to the original intent of the first edition, fundamental object-oriented concepts. I like to think that whatever success the first edition of the book had was because it focused on fundamental object-oriented concepts. In some ways we have gone full circle because this edition encapsulates all the technologies mentioned above.

Finally, the concepts that ultimately encapsulate these technologies into a design methodology are represented by SOLID, which is woven into all the chapters of this edition as well as two new chapters at the end of the book.

The five SOLID principles are

- **SRP**—Single Responsibility Principle
- **OCP**—Open/Close Principle
- **LSP**—Liskov Substitution Principle
- **IPS**—Interface Segregation Principle

- **DIP**—Dependency Inversion Principle

I often think of the first nine chapters as representing what I consider classical object-oriented principles. The last three chapters on design patterns, avoiding dependencies, and SOLID build on the classical principles and present a strong methodology.

The Intended Audience

This book is a general introduction to the concepts of object-oriented programming. The term *concepts* is important because, while code is certainly used to reinforce the topics covered, the primary focus of this book is to ground the reader in the object-oriented thought process. It is also important for programmers to understand that OOP does not represent a distinct paradigm (as many believe)—OOP is simply one part of a vast toolkit available to modern software developers.

When the material for the first edition of this book was initially created in 1995, OOP was in its infancy. I can say this because, other than pockets of OO languages such as Smalltalk, there really were no true object-oriented languages in play at the time. C++, which does not enforce object-oriented constructs, was the dominant C-based language. Java 1.0 was released in 1996 and C# 1.0 in 2002. In fact, when the first edition of this book was published in 1999, there was no certainty that OO would actually become the leading development paradigm. (Java 2 wasn't even released until December 1998.) Despite its current dominance, there are some interesting chinks in the OOP armor to be addressed.

Thus, the audience for the first edition differs from the audience today.

From 1995 until as late as 2010, I was basically retraining many structured programmers in the art of OOP. The vast majority of these students had grown up with COBOL, FORTRAN, C, and VB, both in college and on the job. Today, students graduating college, writing video games, creating websites, or producing mobile apps have almost certainly learned programming using an object-oriented language. Thus, the approach of the fifth edition of this book is significantly different from the first edition, or second, etc. Rather than teaching structured programmers to become OO

developers, we are now teaching programmers who have grown up with OO languages.

The intended audience for this book includes business managers, designers, developers, programmers, and project managers: in short, anyone who wants to gain a general understanding of what object orientation is all about. My hope is that reading this book will provide a strong foundation for moving to other books covering more advanced topics.

The Book's Approach

It should be obvious by now that I am a firm believer in becoming comfortable with the object-oriented thought process before jumping into a programming language or modeling language. This book is filled with examples of code and UML class diagrams; however, you do not need to know a specific programming language or UML to read it. After all I have said about learning the concepts first, why is there so much code and class diagrams?

First, code and class diagrams are great for illustrating OO concepts. Second, they are integral to the OO process and should be addressed at an introductory level. The key is not to focus on Java, C#, and so on but to use them as aids in the understanding of the underlying concepts.

Note that I really like using UML class diagrams as a visual aid to illustrate classes, and their attributes and methods. In fact, the class diagrams are the only component of UML used in this book. I believe that the UML class diagrams offer a great way to model the conceptual nature of object models. I continue to use object models as an educational tool to illustrate class design and how classes relate to one another.

The code examples in the book illustrate concepts such as loops and functions; however, understanding the code itself is not a prerequisite for understanding the concepts. It might be helpful
to have a book at hand that covers specific languages' syntax if you want to get more detailed.

I cannot state too strongly that this book does *not* teach Java, C# .NET, VB .NET, Objective-C, Swift, or UML, all of which can command volumes unto themselves. It is also important to understand that this is a book of concepts, and the intent of the examples in this book is not, necessarily, to describe the *optimal* way to design your classes; they are an educational exercise meant to get you thinking about OO concepts. For example, it is obvious that you won't create many models using penguins and barkless dogs on the job—but using them is a fun way to demonstrate the concepts. With all of this in mind, it is my hope that this book will whet your appetite for other OO topics, such as OO analysis, object-oriented design, and OO programming.

Source Code Used in This Book

The sample code described throughout this book is available on the publisher's website. Go to informit.com/register and register your book for access to downloads.

1. Introduction to Object-Oriented Concepts

Although many programmers don't realize it, object-oriented (OO) software development has been around since the early 1960s. It wasn't until the mid to late 1990s that the object-oriented paradigm started to gain momentum, despite the fact that popular object-oriented programming languages such as Smalltalk and C++ were already widely used.

The rise of OO methodologies coincides with the emergence of the Internet as a business and entertainment platform. In short, objects work well over a network. And after it became obvious that the Internet was here to stay, object-oriented technologies were already well positioned to develop the new web-based technologies.

It is important to note that the title of this first chapter is "Introduction to Object-Oriented Concepts." The operative word here is "concepts" and not "technologies." Technologies change very quickly in the software industry, whereas concepts evolve. I use the term "evolve" because, although they remain relatively stable, they do change. And this is what is really cool about focusing on the concepts. Despite their consistency, they are always undergoing reinterpretations, and this allows for some very interesting discussions.

This evolution can be easily traced over the past 25 years or so as we follow the progression of the various industry technologies from the first primitive browsers of the mid to late 1990s to the mobile/phone/web applications that dominate today. As always, new developments are just around the corner as we explore hybrid apps and more. Throughout this journey, OO concepts have been there every step of the way. That is why the topics of this chapter are so important. These concepts are just as relevant today as they were 25 years ago.

The Fundamental Concepts

The primary point of this book is to get you thinking about how the concepts are used in designing object-oriented systems. Historically, object-oriented languages are defined by the following: encapsulation, inheritance, and polymorphism (what I call "classical" OO). Thus, if a language does not implement all of these, it is generally not considered completely object-oriented. Along with these three terms, I always include composition in the mix; thus, my list of object-oriented concepts looks like this:

- Encapsulation
- Inheritance
- Polymorphism
- Composition

We will discuss all these in detail as we proceed through the rest of the book.

One of the issues that I have struggled with right from the first edition of this book is how these concepts relate directly to current design practices, which are always changing. For example, there has always been debate about using inheritance in an OO design. Does inheritance actually break encapsulation? (This topic will be covered in later chapters.) Even now, many developers try to avoid inheritance as much as possible. So this raises the question: Should inheritance be used at all?

My approach is, as always, to stick to concepts. Whether or not you use inheritance, you at least need to understand what inheritance is, thus enabling you to make an educated design choice. It is important not to forget that inheritance will almost certainly be encountered in code maintenance, so you need to learn it regardless.

As mentioned in the introduction, the intended audience is those who want a general introduction to fundamental OO concepts. With this statement in mind, in this chapter I present the fundamental object-oriented concepts with the hope that you will then gain a solid foundation for making important design decisions. The concepts covered here touch on most, if not all, of the

topics covered in subsequent chapters, which explore these issues in much greater detail.

Objects and Legacy Systems

As OO moved into the mainstream, one of the issues facing developers was the integration of new OO technologies with existing systems. Lines were being drawn between OO and structured (or procedural) programming, which was the dominant development paradigm at the time. I always found this odd because, in my mind, object-oriented and structured programming do not compete with each other. They are complementary because objects integrate well with structured code. Even now, I often hear this question: are you a structured programmer or an object-oriented programmer? Without hesitation, I would answer: both.

In the same vein, object-oriented code is not meant to replace structured code. Many non-OO *legacy systems* (that is, older systems that are already in place) are doing the job quite well, so why risk potential disaster by changing or replacing them? In most cases you should not change them, at least not for the sake of change. There is nothing inherently wrong with systems written in non-OO code. However, brand-new development definitely warrants the consideration of using OO technologies (in some cases, there is no choice but to do so).

Although there has been a steady and significant growth in OO development in the past 25 years, the global community's dependence on networks such as the Internet and mobile infrastructures has helped catapult it even further into the mainstream. The explosion of transactions performed on browsers and mobile apps has opened up brand-new markets, where much of the software development is new and mostly unencumbered by legacy concerns. Even when there are legacy concerns, there is a trend to wrap the legacy systems in object wrappers.

> ### Object Wrappers
>
> Object wrappers are object-oriented code that includes other code inside. For example, you can take structured code (such as loops and conditions) and *wrap* it inside an object to make it look like an

> object. You can also use object wrappers to *wrap* functionality such as security features, nonportable hardware features, and so on. Wrapping structured code is covered in detail in Chapter 6, "Designing with Objects."

One of the most interesting areas of software development is the integration of legacy code with mobile- and web-based systems. In many cases, a mobile web front end ultimately connects to data that resides on a mainframe. Developers who can combine the skills of mainframe and mobile web development are in demand.

You probably experience objects in your daily life without even realizing it. These experiences can take place in your car, when you're talking on your cell phone, using your home entertainment system, playing computer games, and many other situations. The electronic highway has, in essence, become an object-based highway. As businesses gravitate toward the mobile web, they are gravitating toward objects because the technologies used for electronic commerce are mostly OO in nature.

> ### Mobile Web
>
> No doubt, the emergence of the Internet provided a major impetus for the shift to object-oriented technologies. This is because objects are well suited for use on networks. Although the Internet was at the forefront of this paradigm shift, mobile networks have now joined the mix in a major way. In this book, the term *mobile web* will be used in the context of concepts that pertain to both mobile app development and web development. The term *hybrid* app is sometimes used to refer to applications that render in browsers on both web and mobile devices.

Procedural Versus OO Programming

Before we delve deeper into the advantages of OO development, let's consider a more fundamental question: What exactly is an object? This is both a complex and a simple question. It is complex because learning any method of software development is not trivial. It is simple because people already think in terms of objects.

> **TIP**
>
> In watching a YouTube video lecture presented by OO guru Robert Martin, his view is that the statement that "people think in terms of objects" was coined by marketing people. Just some food for thought.

For example, when you look at a person, you see the person as an object. And an object is defined by two components: attributes and behaviors. A person has attributes, such as eye color, age, height, and so on. A person also has behaviors, such as walking, talking, breathing, and so on. In its basic definition, an *object* is an entity that contains *both* data and behavior. The word *both* is the key difference between OO programming and other programming methodologies. In procedural programming, for example, code is placed into totally distinct functions or procedures. Ideally, as shown in Figure 1.1, these procedures then become "black boxes," where inputs go in and outputs come out. Data is placed into separate structures and is manipulated by these functions or procedures.

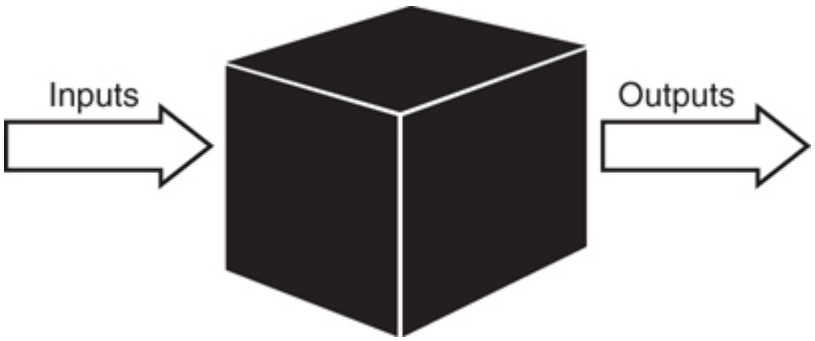

Figure 1.1 Black boxes.

> **Difference Between OO and Procedural**
>
> In OO design, the attributes and behaviors are contained within a single object, whereas in procedural, or structured, design the attributes and behaviors are normally separated.

As OO design grew in popularity, one of the realities that initially slowed its acceptance was that there were a lot of non-OO systems in place that worked perfectly fine. Thus, it did not make any business sense to change the systems for the sake of change. Anyone who is familiar with any computer system

knows that any change can spell disaster—even if the change is perceived to be slight.

This situation came into play with the lack of acceptance of OO databases. At one point in the emergence of OO development, it seemed somewhat likely that OO databases would replace relational databases. However, this never happened. Businesses have a lot of money invested in relational databases, and one overriding factor discouraged conversion: they worked. When all the costs and risks of converting systems from relational to OO databases became apparent, there was no compelling reason to switch.

In fact, the business forces have now found a happy middle ground. Much of the software development practices today have flavors of several development methodologies, such as OO and structured.

As illustrated in Figure 1.2, in structured programming the data is often separated from the procedures, and often the data is global, so it is easy to modify data that is outside the scope of your code. This means that access to data is uncontrolled and unpredictable (that is, multiple functions may have access to the global data). Second, because you have no control over who has access to the data, testing and debugging are much more difficult. Objects address these problems by combining data and behavior into a nice, complete package.

Figure 1.2 Using global data.

> ### Proper Design
>
> We can state that when properly designed, there is no such thing as global data in an OO model. This fact provides a high amount of data integrity in OO systems.

Rather than replacing other software development paradigms, objects are an evolutionary response. Structured programs have complex data structures, such as arrays, and so on. C++ has structures, which have many of the characteristics of objects (classes).

However, objects are much more than data structures and primitive data types, such as integers and strings. Although objects do contain entities such as integers and strings, which are used to represent attributes, they also contain methods, which represent behaviors. In an object, methods are used

to perform operations on the data as well as other actions. Perhaps more important, you can control access to members of an object (both attributes and methods). This means that some members, both attributes and methods, can be hidden from other objects. For instance, an object called `Math` might contain two integers, called `myInt1` and `myInt2`. Most likely, the `Math` object also contains the necessary methods to set and retrieve the values of `myInt1` and `myInt2`. It might also contain a method called `sum()` to add the two integers together.

> ### Data Hiding
>
> In OO terminology, data are referred to as *attributes*, and behaviors are referred to as methods. Restricting access to certain attributes and/or methods is called data *hiding*.

By combining the attributes and methods in the same entity, which in OO parlance is called *encapsulation*, we can control access to the data in the `Math` object. By defining these integers as off-limits, another logically unconnected function cannot manipulate the integers `myInt1` and `myInt2`—only the `Math` object can do that.

> ### Sound Class Design Guidelines
>
> Keep in mind that it is possible to create poorly designed OO classes that do not restrict access to class attributes. The bottom line is that you can design bad code just as efficiently with OO design as with any other programming methodology. Simply take care to adhere to sound class design guidelines (see Chapter 5, "Class Design Guidelines").

What happens when another object—for example, `myObject`—wants to gain access to the sum of `myInt1` and `myInt2`? It asks the `Math` object: `myObject` sends a message to the `Math` object. Figure 1.3 shows how the two objects communicate with each other via their methods. The message is really a call to the `Math` object's `sum` method. The `sum` method then returns the value to `myObject`. The beauty of this is that `myObject` does not need to know how the sum is calculated (although I'm sure it can guess). With this design methodology in place, you can change how the `Math` object

calculates the sum without making a change to `myObject` (as long as the means to retrieve the sum do not change). All you want is the sum—you *don't care* how it is calculated.

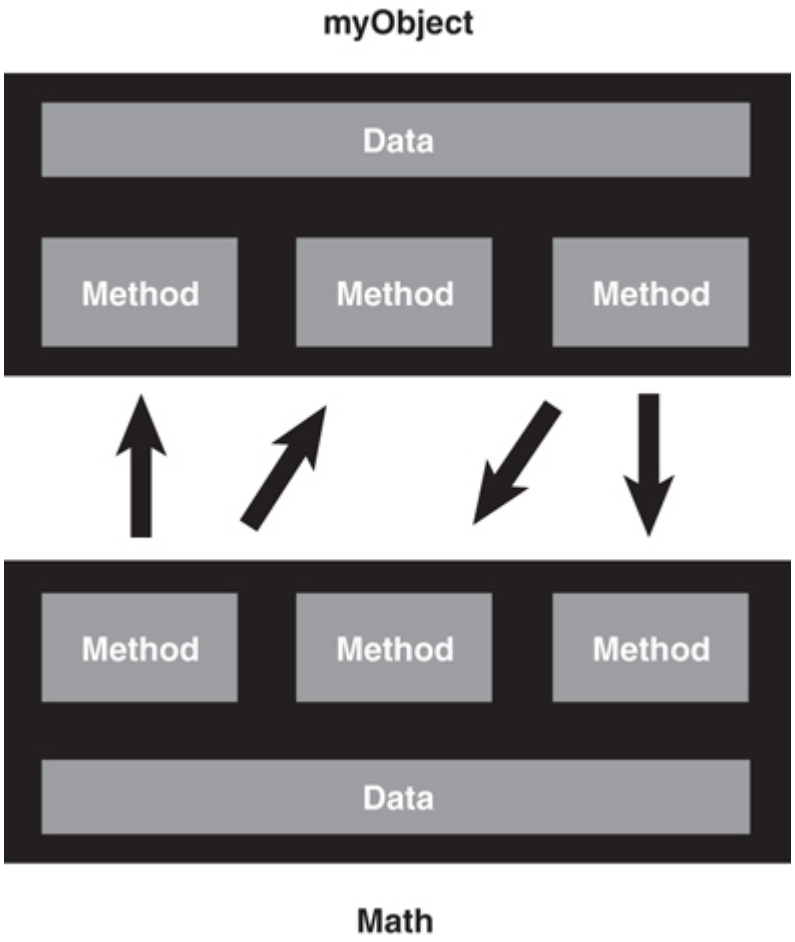

Figure 1.3 Object-to-object communication.

Using a simple calculator example illustrates this concept. When determining a sum with a calculator, all you use is the calculator's interface—the keypad and LED display. The calculator has a sum method that is invoked when you press the correct key sequence. You may get the correct answer back; however, you have no idea how the result was obtained—either electronically or algorithmically.

Calculating the sum is not the responsibility of `myObject`—it's the `Math` object's responsibility. As long as `myObject` has access to the `Math` object, it can send the appropriate messages and obtain the requested result. In general, objects should not manipulate the internal data of other objects (that is, `myObject` should not directly change the value of `myInt1` and

`myInt2`). And, for reasons we will explore later, it is normally better to build small objects with specific tasks rather than build large objects that perform many.

Moving from Procedural to Object-Oriented Development

Now that we have a general understanding about some of the differences between procedural and object-oriented technologies, let's delve a bit deeper into both.

Procedural Programming

Procedural programming normally separates the data of a system from the operations that manipulate the data. For example, if you want to send information across a network, only the relevant data is sent (see Figure 1.4), with the expectation that the program at the other end of the network pipe knows what to do with it. In other words, some sort of handshaking agreement must be in place between the client and the server to transmit the data. In this model, it is quite possible that no code is actually sent over the wire.

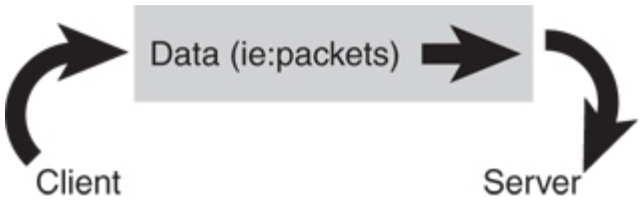

Figure 1.4 Data transmitted over a wire.

OO Programming

The fundamental advantage of OO programming is that the data and the operations that manipulate the data (the code) are both encapsulated in the object. For example, when an object is transported across a network, the entire object, including the data and behavior, goes with it.

> ### A Single Entity

> Although thinking in terms of a single entity is great in theory, in many cases, the behaviors themselves may not be sent because both sides have copies of the code. However, it is important to think in terms of the entire object being sent across the network as a single entity.

In Figure 1.5, the `Employee` object is sent over the network.

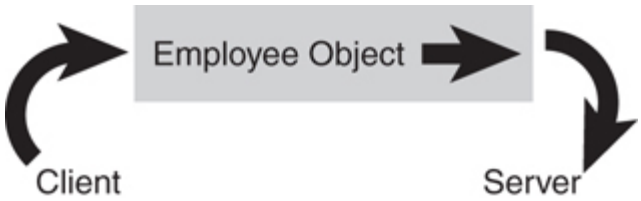

Figure 1.5 Objects transmitted over a wire.

> **Proper Design**
>
> A good example of this concept is an object that is loaded by a browser. Often, the browser has no idea of what the object will do ahead of time because the code is not there previously. When the object is loaded, the browser executes the code within the object and uses the data contained within the object.

What Exactly Is an Object?

Objects are the building blocks of an OO program. A program that uses OO technology is basically a collection of objects. To illustrate, let's consider that a corporate system contains objects that represent employees of that company. Each of these objects is made up of the data and behavior described in the following sections.

Object Data

The data stored within an object represents the state of the object. In OO programming terminology, this data is called *attributes*. In our example, as shown in Figure 1.6, employee attributes could be Social Security numbers, date of birth, gender, phone number, and so on. The attributes contain the information that differentiates between the various objects, in this case the

employees. Attributes are covered in more detail later in this chapter in the discussion on classes.

Figure 1.6 Employee attributes.

Object Behaviors

The *behavior* of an object represents what the object can do. In procedural languages the behavior is defined by procedures, functions, and subroutines. In OO programming terminology, these behaviors are contained in *methods*, and you invoke a method by sending a message to it. In our employee example, consider that one of the behaviors required of an employee object is to set and return the values of the various attributes. Thus, each attribute would have corresponding methods, such as `setGender()` and `getGender()`. In this case, when another object needs this information, it can send a message to an employee object and ask it what its gender is.

Not surprisingly, the application of getters and setters, as with much of object-oriented technology, has evolved since the first edition of this book was published. This is especially true when it comes to data. Remember that one of the most interesting, not to mention powerful, advantages of using objects is that the data is part of the package—it is not separated from the code.

The emergence of XML has not only focused attention on presenting data in a portable manner; it also has facilitated alternative ways for the code to access the data. In .NET techniques, the getters and setters are considered properties of the data itself.

For example, consider an attribute called `Name`, using Java, that looks like the following:

```
public String Name;
```

The corresponding getter and setter would look like this:

```
public void setName (String n) {name = n;};
public String getName() {return name;};
```

Now, when creating an XML attribute called `Name`, the definition in C# .NET may look something like this, although you can certainly use the same approach as the Java example:

```
private string strName;

public String Name
{
    get { return this.strName; }
    set {
        if (value == null) return;
        this.strName = value;
    }
}
```

In this technique, the getters and setters are actually *properties* of the attributes—in this case, `Name`.

Regardless of the approach, the purpose is the same—controlled access to the attribute. For this chapter, I want to first concentrate on the conceptual nature of accessor methods; we will get more into properties in later chapters.

> ### Getters and Setters
>
> The concept of getters and setters supports the concept of data hiding. Because other objects should not directly manipulate data within another object, the getters and setters provide controlled

> access to an object's data. Getters and setters are sometimes called accessor methods and mutator methods, respectively.

Note that we are showing only the interface of the methods, and not the implementation. The following information is all the user needs to know to effectively use the methods:

- The name of the method
- The parameters passed to the method
- The return type of the method

To illustrate behaviors, consider Figure 1.7.

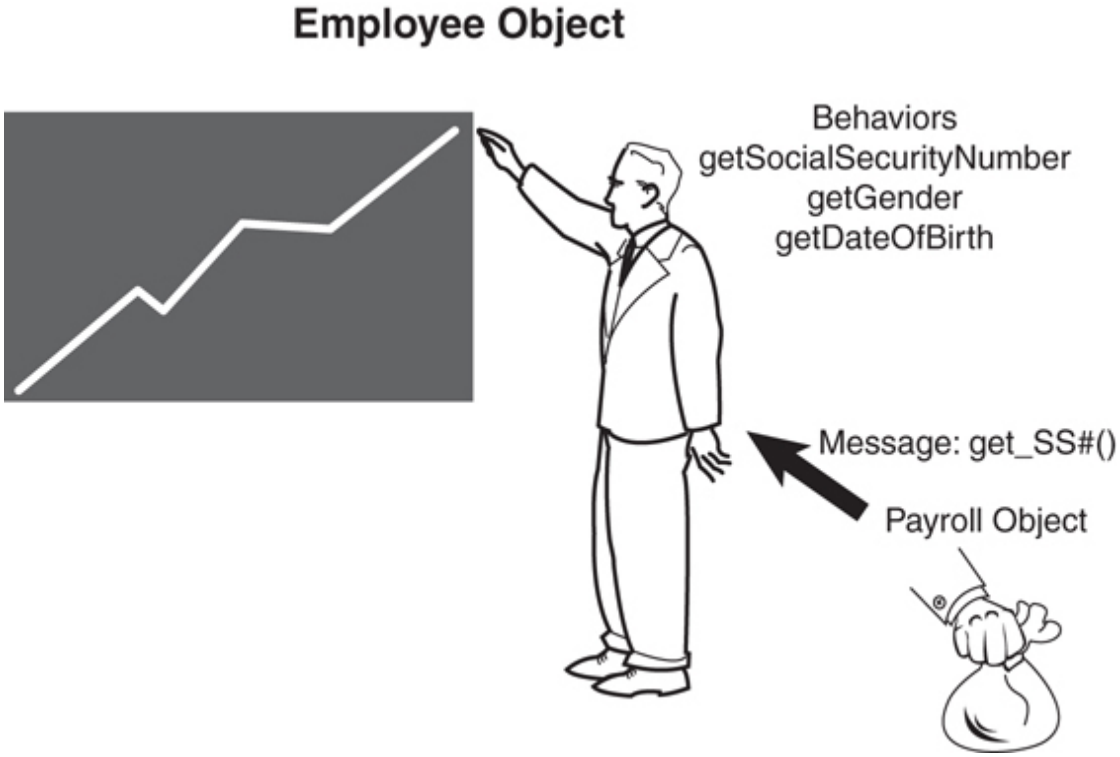

Figure 1.7 Employee behaviors.

In Figure 1.7, the `Payroll` object contains a method called `CalculatePay()` that calculates the pay for a specific employee. Among other information, the `Payroll` object must obtain the Social Security number of this employee. To get this information, the payroll object must send a message to the `Employee` object (in this case, the `getSocialSecurityNumber()` method). Basically, this means that the

`Payroll` object calls the `getSocialSecurityNumber()` method of the `Employee` object. The employee object recognizes the message and returns the requested information.

To illustrate further, Figure 1.8 is a class diagram representing the `Employee/Payroll` system we have been talking about.

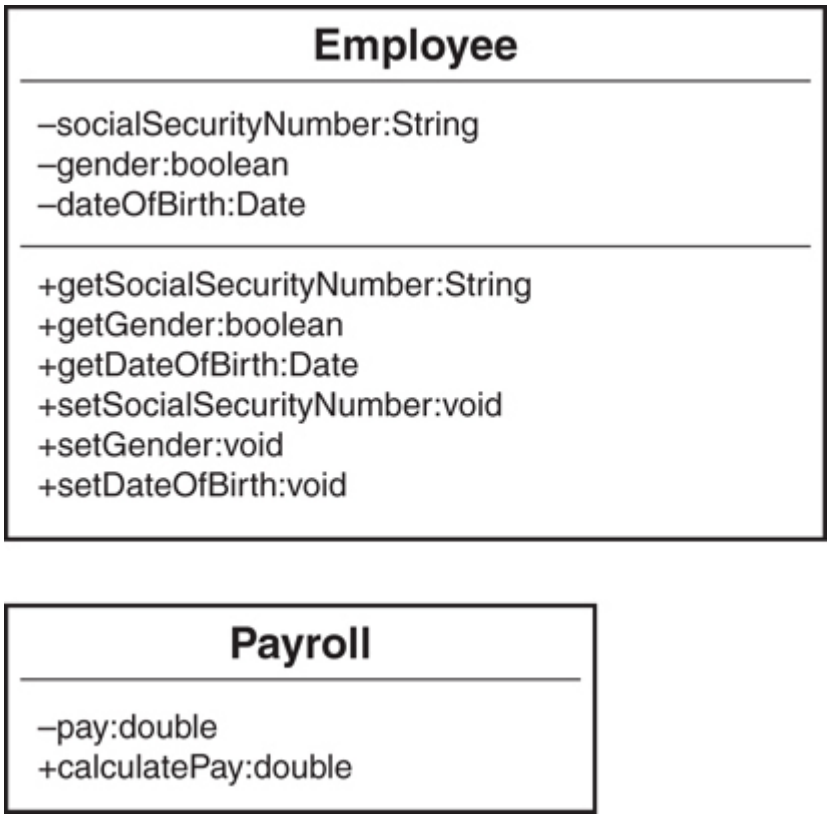

Figure 1.8 Employee and payroll class diagrams.

> ### UML Class Diagrams
>
> Because this is the first class diagram we have seen, it is very basic and lacks some of the constructs (such as constructors) that a proper class should contain. Fear not—we will discuss class diagrams and constructors in more detail in Chapter 3, "More Object-Oriented Concepts."

Each class diagram is defined by three separate sections: the name itself, the data (attributes), and the behaviors (methods). In Figure 1.8, the `Employee` class diagram's attribute section contains `SocialSecurityNumber`, `Gender`, and `DateofBirth`, whereas the method section contains the

methods that operate on these attributes. You can use UML modeling tools to create and maintain class diagrams that correspond to real code.

> ### Modeling Tools
>
> Visual modeling tools provide a mechanism to create and manipulate class diagrams using the Unified Modeling Language (UML). Class diagrams are used and discussed throughout this book. They are used as a tool to help visualize classes and their relationships to other classes. The use of UML in this book is limited to class diagrams.

We will get into the relationships between classes and objects later in this chapter, but for now you can think of a class as a template from which objects are made. When an object is created, we say that the objects are instantiated. Thus, if we create three employees, we are actually creating three totally distinct instances of an `Employee` class. Each object contains its own copy of the attributes and methods. For example, consider Figure 1.9. An employee object called `John` (John is its identity) has its own copy of all the attributes and methods defined in the `Employee` class. An employee object called `Mary` has its own copy of attributes and methods. They both have a separate copy of the `DateOfBirth` attribute and the `getDateOfBirth` method.

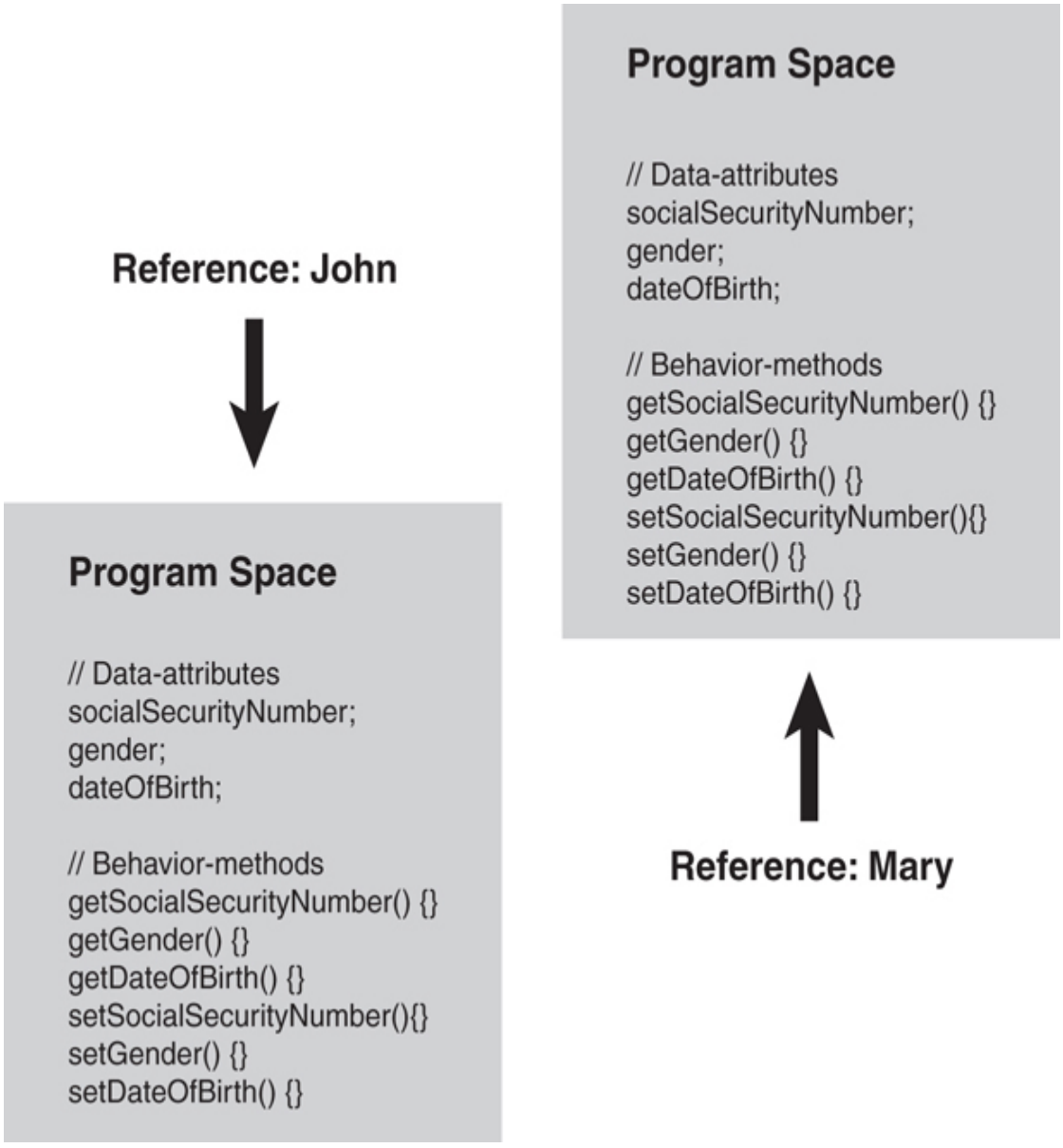

Figure 1.9 Program spaces.

An Implementation Issue

Be aware that there is not necessarily a physical copy of each method for each object. Rather, each object points to the same implementation. However, this is an issue left up to the compiler/operating platform. From a conceptual level, you can think of objects as being wholly independent and having their own attributes and methods.

What Exactly Is a Class?

In short, a class is a blueprint for an object. When you instantiate an object, you use a class as the basis for how the object is built. In fact, trying to explain classes and objects is really a chicken-and-egg dilemma. It is difficult to describe a class without using the term *object* and vice versa. For example, a specific individual bike is an object. However, someone had to have created the blueprints (that is, the class) to build the bike. In OO software, unlike the chicken-and-egg dilemma, we do know what comes first—the class. An object cannot be instantiated without a class. Thus, many of the concepts in this section are similar to those presented earlier in the chapter, especially when we talk about attributes and methods.

Although this book focuses on the concepts of OO software and not on a specific implementation, it is often helpful to use code examples to explain some concepts, so Java code fragments are used throughout the book to help explain some concepts when appropriate. However, for certain key examples, the code is provided in several languages as downloads.

The following sections describe some of the fundamental concepts of classes and how they interact.

Creating Objects

Classes can be thought of as the templates, or cookie cutters, for objects as seen in Figure 1.10. A class is used to create an object.

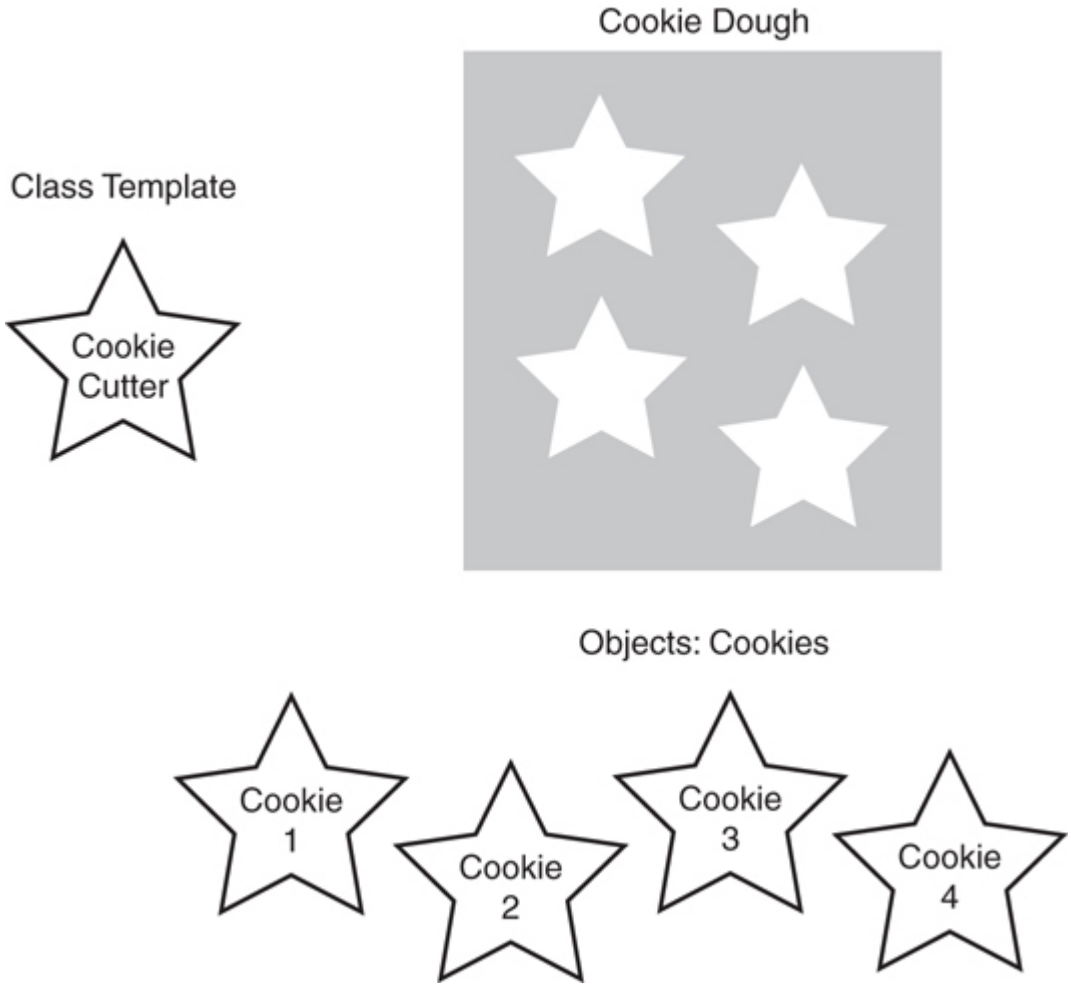

Figure 1.10 Class template.

A class can be thought of as a sort of higher-level data type. For example, just as you create an integer or a float:

```
int x;
float y;
```

you can also create an object by using a predefined class:

```
myClassmyObject;
```

In this example, the names themselves make it obvious that `myClass` is the class and `myObject` is the object.

Remember that each object has its own attributes (data) and behaviors (functions or routines). A class defines the attributes and behaviors that all objects created with this class will possess. Classes are pieces of code.

Objects instantiated from classes can be distributed individually or as part of a library. Because objects are created from classes, it follows that classes must define the basic building blocks of objects (attributes, behavior, and messages). In short, you must design a class before you can create an object.

For example, here is a definition of a `Person` class:

```
public class Person{

    //Attributes
    private String name;
    private String address;

    //Methods
    public String getName(){
        return name;
    }
    public void set        Name(String n){
        name = n;
    }

    public String getAddress(){
        return address;
    }
    public void setAddress(String adr){
        address = adr;
    }

}
```

Attributes

As you already saw, the data of a class is represented by attributes. Each class must define the attributes that will store the state of each object instantiated from that class. In the `Person` class example in the previous section, the `Person` class defines attributes for `name` and `address`.

> ### Access Designations
>
> When a data type or method is defined as public, other objects can directly access it. When a data type or method is defined as private, only that specific object can access it. Another access modifier,

> protected, allows access by related objects, which you'll learn about in Chapter 3.

Methods

As you learned earlier in the chapter, methods implement the required behavior of a class. Every object instantiated from this class includes methods as defined by the class. Methods may implement behaviors that are called from other objects (messages) or provide the fundamental, internal behavior of the class. Internal behaviors are private methods that are not accessible by other objects. In the `Person` class, the behaviors are `getName()`, `setName()`, `getAddress()`, and `setAddress()`. These methods allow other objects to inspect and change the values of the object's attributes. This is a common technique in OO systems. In all cases, access to attributes within an object should be controlled by the object itself —no other object should directly change an attribute of another.

Messages

Messages are the communication mechanism between objects. For example, when Object A invokes a method of Object B, Object A is sending a message to Object B. Object B's response is defined by its return value. Only the public methods, not the private methods, of an object can be invoked by another object. The following code illustrates this concept:

```
public class Payroll{

    String name;
    Person p = new Person();
    p.setName("Joe");

        ... code

    name = p.getName();
}
```

In this example (assuming that a `Payroll` object is instantiated), the `Payroll` object is sending a message to a `Person` object, with the purpose of retrieving the name via the `getName()` method. Again, don't

worry too much about the actual code, because we are really interested in the concepts. We address the code in detail as we progress through the book.

Using Class Diagrams as a Visual Tool

Over the years, many tools and modeling methodologies have been developed to assist in designing software systems. Right from the start, I have used UML class diagrams to assist in the educational process. Although it is beyond the scope of this book to describe UML in any detail, we will use UML class diagrams to illustrate the classes that we build. In fact, we have already used class diagrams in this chapter. Figure 1.11 shows the `Person` class diagram we discussed earlier in the chapter.

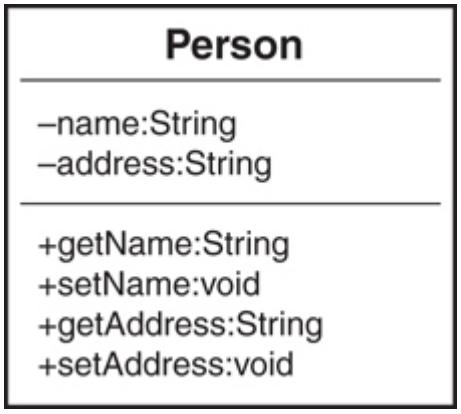

Figure 1.11 The `Person` class diagram.

As we saw previously, notice that the attributes and methods are separated (the attributes on the top and the methods on the bottom). As we delve more deeply into OO design, these class diagrams will get much more sophisticated and convey much more information on how the different classes interact with each other.

Encapsulation and Data Hiding

One of the primary advantages of using objects is that the object need not reveal all its attributes and behaviors. In good OO design (at least what is generally accepted as good), an object should reveal only the interfaces that other objects must have to interact with it. Details not pertinent to the use of

the object should be hidden from all other objects—basically a "need to know" basis.

Encapsulation is defined by the fact that objects contain both the attributes and behaviors. Data hiding is a major part of encapsulation.

For example, an object that calculates the square of a number must provide an interface to obtain the result. However, the internal attributes and algorithms used to calculate the square need not be made available to the requesting object. Robust classes are designed with encapsulation in mind. In the next sections, we cover the concepts of interface and implementation, which are the basis of encapsulation.

Interfaces

We have seen that the interface defines the fundamental means of communication between objects. Each class design specifies the interfaces for the proper instantiation and operation of objects. Any behavior that the object provides must be invoked by a message sent using one of the provided interfaces. The interface should completely describe how users of the class interact with the class. In most OO languages, the methods that are part of the interface are designated as `public`.

> **Private Data**
>
> For data hiding to work properly, all attributes should be declared as private. Thus, attributes are never part of the interface. Only the public methods are part of the class interface. Declaring an attribute as public breaks the concept of data hiding.

Let's look at the example just mentioned: calculating the square of a number. In this example, the interface would consist of two pieces:

- How to instantiate a `Square` object
- How to send a value to the object and get the square of that value in return

As discussed earlier in the chapter, if a user needs access to an attribute, a method is created to return the value of the attribute (a getter). If a user then wants to obtain the value of an attribute, a method is called to return its value. In this way, the object that contains the attribute controls access to it. This is of vital importance, especially in security, testing, and maintenance. If you control the access to the attribute, when a problem arises, you do not have to worry about tracking down every piece of code that might have changed the attribute—it can be changed in only one place (the setter).

From a security perspective, you don't want uncontrolled code to change or retrieve sensitive data. For example, when you use an ATM, access to data is controlled by asking for a PIN.

> ### Signatures—Interfaces Versus Interfaces
>
> Don't confuse the interfaces used to extend classes with the interface of a class. I like to equate the interfaces, represented by methods, as "signatures."

Implementations

Only the public attributes and methods are considered the interface. The user should not see any part of the internal implementation, interacting with an object solely through class interfaces. Thus, anything defined as private is inaccessible to the user and considered part of the class's internal implementation.

In the previous example, for instance the `Employee` class, only the attributes were hidden. In many cases, there will be methods that also should be hidden and thus not part of the interface. Continuing the example of the square root from the previous section, the user does not care how the square root is calculated—as long as it is the correct answer. Thus, the implementation can change, and it will not affect the user's code. For example, the company that produces the calculator can change the algorithm (perhaps because it is more efficient) without affecting the result.

A Real-World Example of the Interface/Implementation Paradigm

Figure 1.12 illustrates the interface/implementation paradigm using real-world objects rather than code. The toaster requires electricity. To get this electricity, the cord from the toaster must be plugged into the electrical outlet, which is the interface. All the toaster needs to do to obtain the required electricity is to implement a cord that complies with the electrical outlet specifications; this is the interface between the toaster and the power company (actually the power industry). That the actual implementation is a coal-powered electric plant is not the concern of the toaster. In fact, for all the toaster cares, the implementation could be a nuclear power plant or a local power generator. With this model, any appliance can get electricity, as long as it conforms to the interface specification as shown in Figure 1.12.

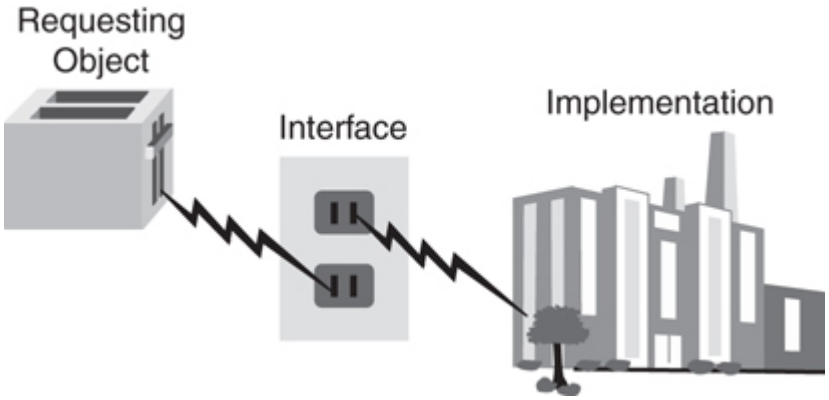

Figure 1.12 Power plant example.

A Model of the Interface/Implementation Paradigm

Let's explore the `Square` class further. Assume that you are writing a class that calculates the squares of integers. You must provide a separate interface and implementation. That is, you must specify a way for the user to invoke and obtain the square value. You must also provide the implementation that calculates the square; however, the user should not know anything about the specific implementation. Figure 1.13 shows one way to do this. Note that in the class diagram, the plus sign (+) designates public and the minus sign (-) designates private. Thus, you can identify the interface by the methods, prefaced with plus signs.

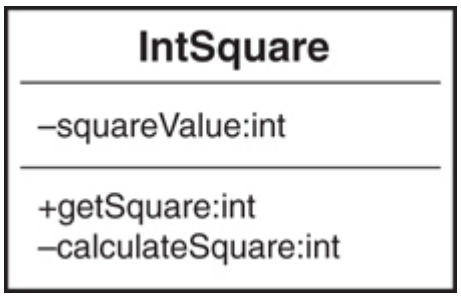

Figure 1.13 The `Square` class.

This class diagram corresponds to the following code:

Click here to view code image

```
public class IntSquare {

    // private attribute
    private int squareValue;

    // public interface
    public in getSquare (int value) {

    squareValue = calculateSquare(value);

        return squareValue;

    }

    // private implementation
    private in calculateSquare (int value) {

        return value*value;

    }
}
```

Note that the only part of the class that the user has access to is the public method `getSquare`, which is the interface. The implementation of the square algorithm is in the method `calculateSquare`, which is private. Also notice that the attribute `SquareValue` is private because users do not need to know that this attribute exists. Therefore, we have hidden the part of the implementation: The object reveals only the interfaces the user needs to interact with it, and details that are not pertinent to the use of the object are hidden from other objects.

If the implementation were to change—suppose you wanted to use the language's built-in square function—you would not need to change the interface. Here the code uses the Java library method `Math.pow`, which performs the same function, but note that the interface is still `calculateSquare`.

Click here to view code image

```
// private implementation
private intcalculateSquare (int value) {

    return = Math.pow(value,2);

}
```

The user would get the same functionality using the same interface, but the implementation would have changed. This is very important when you're writing code that deals with data; for example, you can move data from a file to a database without forcing the user to change any application code.

Inheritance

Inheritance enables a class to inherit the attributes and methods of another class. This provides the ability to create new classes by abstracting out common attributes and behaviors from another class.

One of the major design issues in OO programming is to factor out commonality of the various classes. For example, suppose you have a `Dog` class and a `Cat` class, and each will have an attribute for eye color. In a procedural model, the code for `Dog` and `Cat` would each contain this attribute. In an OO design, the color attribute could be moved up to a class called `Mammal`—along with any other common attributes and methods. In this case, both `Dog` and `Cat` inherit from the `Mammal` class, as shown in Figure 1.14.

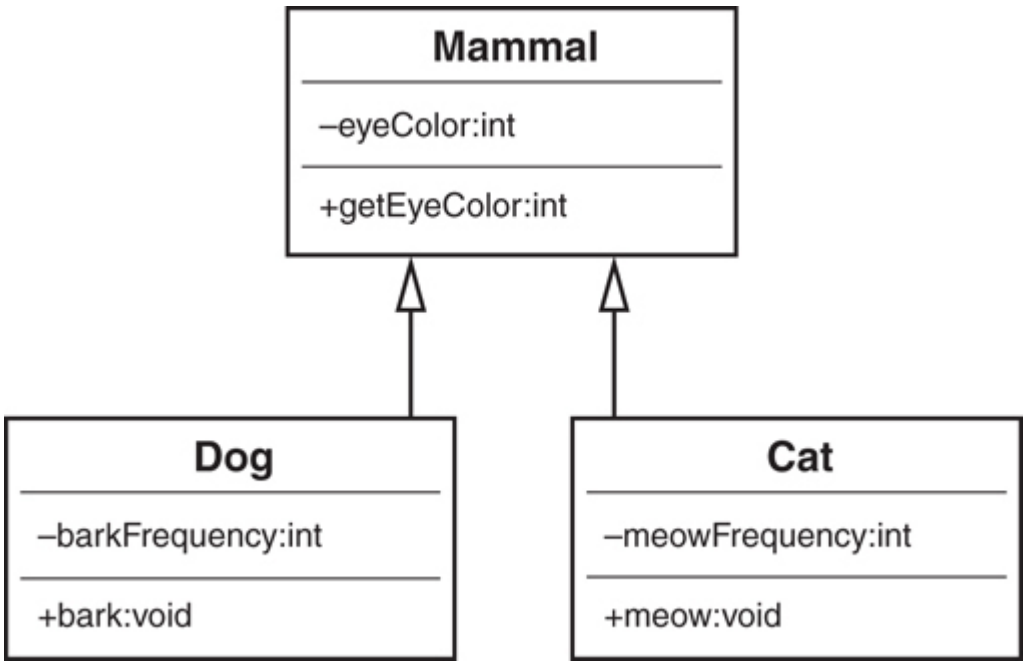

Figure 1.14 Mammal hierarchy.

The `Dog` and `Cat` classes both inherit from `Mammal`. This means that a `Dog` class has the following attributes:

```
eyeColor          // inherited from Mammal
barkFrequency     // defined only for Dogs
```

In the same vein, the `Dog` object has the following methods:

```
getEyeColor       // inherited from Mammal
bark              // defined only for Dogs
```

When the `Dog` or the `Cat` object is instantiated, it contains everything in its own class, as well as everything from the parent class. Thus, `Dog` has all the properties of its class definition, as well as the properties inherited from the `Mammal` class.

> ### Behaviors
>
> It is worth noting that behaviors today tend to be described in interfaces and that inheritance of attributes is the most common use of direct inheritance. In this way, the behaviors are abstracted away from their data.

Superclasses and Subclasses

The superclass, or parent class (sometimes called base class), contains all the attributes and behaviors that are common to classes that inherit from it. For example, in the case of the `Mammal` class, all mammals have similar attributes, such as `eyeColor` and `hairColor`, as well as behaviors, such as `generateInternalHeat` and `growHair`. All mammals have these attributes and behaviors, so it is not necessary to duplicate them down the inheritance tree for each type of mammal. Duplication requires a lot more work, and perhaps more worrisome, it can introduce errors and inconsistencies.

The subclass, or child class (sometimes called derived class) is an extension of the superclass. Thus, the `Dog` and `Cat` classes inherit all those common attributes and behaviors from the `Mammal` class. The `Mammal` class is considered the superclass of the `Dog` and the `Cat` subclasses, or child classes.

Inheritance provides a rich set of design advantages. When you're designing a `Cat` class, the `Mammal` class provides much of the functionality needed. By inheriting from the `Mammal` object, `Cat` already has all the attributes and behaviors that make it a true mammal. To make it more specifically a cat type of mammal, the `Cat` class must include any attributes or behaviors that pertain solely to a cat.

Abstraction

An inheritance tree can grow quite large. When the `Mammal` and `Cat` classes are complete, other mammals, such as dogs (or lions, tigers, and bears), can be added quite easily. The `Cat` class can also be a superclass to other classes. For example, it might be necessary to abstract the `Cat` class further, to provide classes for Persian cats, Siamese cats, and so on. Just as with `Cat`, the `Dog` class can be the parent for `GermanShepherd` and `Poodle` (see Figure 1.15). The power of inheritance lies in its abstraction and organization techniques.

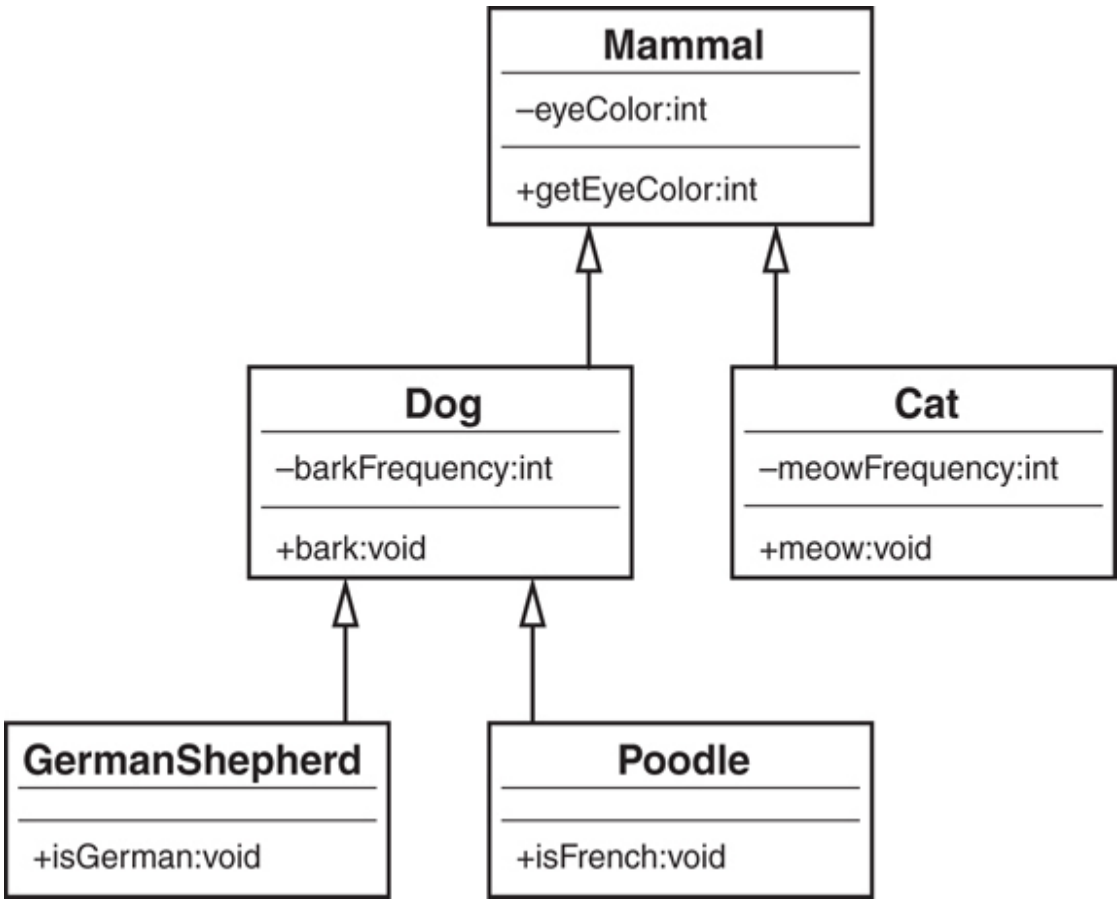

Figure 1.15 Mammal UML diagram.

These multiple levels of abstraction are one of the reasons why many developers are wary of using inheritance at all. As we will see often, it is difficult to decide how much abstraction is required. For example, if a penguin is a bird and a hawk is a bird, should they both inherit from a class called `Bird`—a class that has a fly method?

In most recent OO languages (such as Java, .NET, and Swift), a class can have only a single parent class; however, a class can have many child classes. Some languages, such as C++, can have multiple parents. The former case is called *single inheritance*, and the latter is called *multiple inheritance*.

> ### Multiple Inheritance
>
> Consider a child that inherits from both parents. Which pair of eyes does the child inherit? This is a significant problem when it comes

> to writing a compiler. C++ allows multiple inheritance; many languages do not.

Note that the classes `GermanShepherd` and `Poodle` both inherit from `Dog`—each contains only a single method. However, because they inherit from `Dog`, they also inherit from `Mammal`. Thus, the `GermanShepherd` and `Poodle` classes contain all the attributes and methods included in `Dog` and `Mammal`, as well as their own (see Figure 1.16).

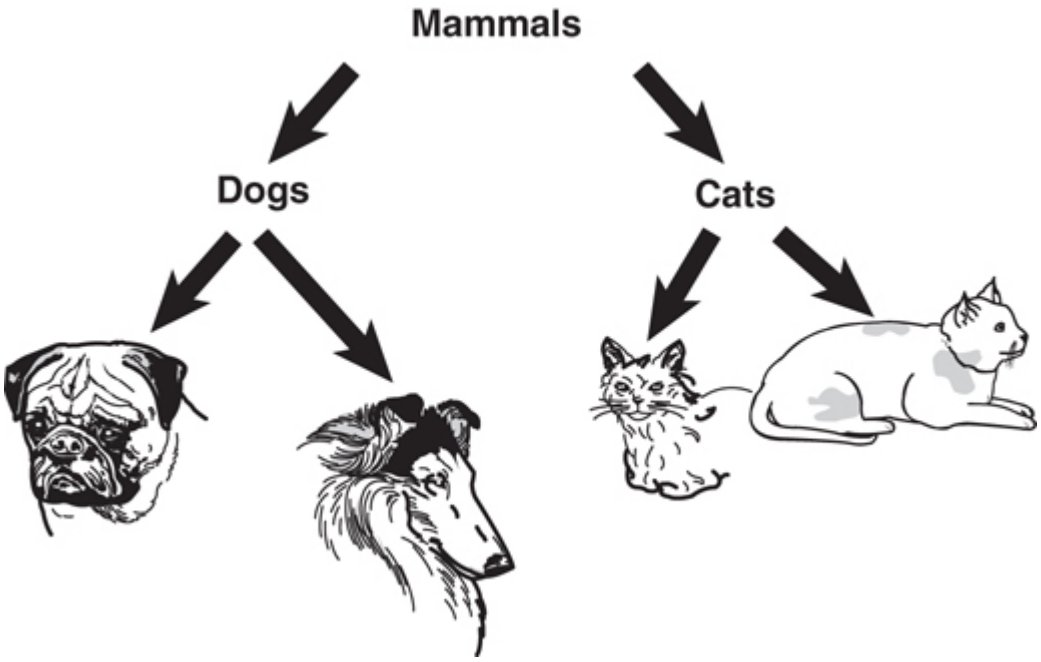

Figure 1.16 Mammal hierarchy.

Is-a Relationships

Consider a `Shape` example where `Circle`, `Square`, and `Star` all inherit directly from `Shape`. This relationship is often referred to as an *is-a relationship* because a circle is a shape, and a square is a shape. When a subclass inherits from a superclass, it can do anything that the superclass can do. Thus, `Circle`, `Square`, and `Star` are all extensions of `Shape`.

In Figure 1.17, the name on each of the objects represents the `draw` method for the `Circle`, `Star`, and `Square` objects, respectively. When we design this `Shape` system, it would be very helpful to standardize how we use the various shapes. Thus, we could decide that if we want to draw a shape, no matter what shape, we will invoke a method called `draw`. If we adhere to

this decision, whenever we want to draw a shape, only the `draw` method needs to be called, regardless of what the shape is. Here lies the fundamental concept of polymorphism—it is the individual object's responsibility, be it a `Circle`, `Star`, or `Square`, to draw itself. This is a common concept in many current software applications, such as drawing and word processing applications.

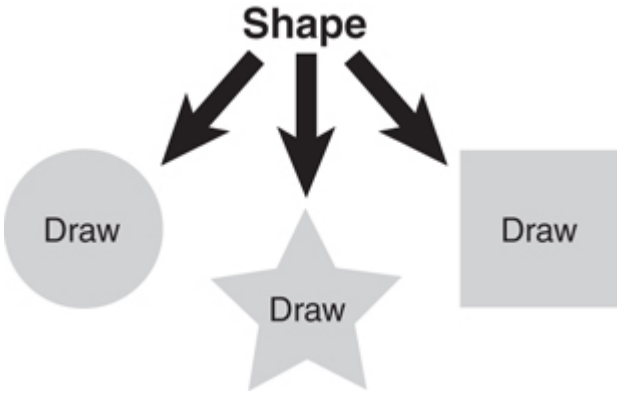

Figure 1.17 The shape hierarchy.

Polymorphism

Polymorphism is a Greek word that literally means many shapes. Although polymorphism is tightly coupled to inheritance, it is often cited separately as one of the most powerful advantages to object-oriented technologies. When a message is sent to an object, the object must have a method defined to respond to that message. In an inheritance hierarchy, all subclasses inherit the interfaces from their superclass. However, because each subclass is a separate entity, each might require a separate response to the same message.

For example, consider the `Shape` class and the behavior called `draw`. When you tell somebody to draw a shape, the first question asked is, "What shape?" No one can draw a shape, because it is an abstract concept (in fact, the `draw` method in the `Shape` code following contains no implementation). You must specify a concrete shape. To do this, you provide the actual implementation in `Circle`. Even though `Shape` has a `draw` method, `Circle` overrides this method and provides its own `draw` method. Overriding basically means replacing an implementation of a parent with one from a child.

For example, suppose you have an array of three shapes—`Circle`, `Square`, and `Star`. Even though you treat them all as `Shape` objects, and send a `draw` message to each `Shape` object, the end result is different for each because `Circle`, `Square`, and `Star` provide the actual implementations. In short, each class is able to respond differently to the same `draw` method and draw itself. This is what is meant by polymorphism.

Consider the following `Shape` class:

```
public abstract class Shape{

    private double area;

    public abstract double getArea();

}
```

The `Shape` class has an attribute called `area` that holds the value for the area of the shape. The method `getArea()` includes an identifier called `abstract`. When a method is defined as `abstract`, a subclass must provide the implementation for this method; in this case, `Shape` is requiring subclasses to provide a `getArea()` implementation. Now let's create a class called `Circle` that inherits from `Shape` (the `extends` keyword specifies that `Circle` inherits from `Shape`):

Click here to view code image

```
public class Circle extends Shape{
    double radius;
    public Circle(double r) {
        radius = r;
    }

    public double getArea() {
        area = 3.14*(radius*radius);
        return (area);
   }

}
```

We introduce a new concept here called a *constructor*. The `Circle` class has a method with the same name, `Circle`. When a method name is the same as the class and no return type is provided, the method is a special

method, called a constructor. Consider a constructor as the entry point for the class, where the object is built; the constructor is a good place to perform initializations and start-up tasks.

The `Circle` constructor accepts a single parameter, representing the radius, and assigns it to the `radius` attribute of the `Circle` class.

The `Circle` class also provides the implementation for the `getArea` method, originally defined as `abstract` in the `Shape` class.

We can create a similar class, called `Rectangle`:

Click here to view code image

```
public class Rectangle extends Shape{

    double length;
    double width;

    public Rectangle(double l, double w){
        length = l;
        width = w;
    }

    public double getArea() {
        area = length*width;
        return (area);
    }

}
```

Now we can create any number of rectangles, circles, and so on and invoke their `getArea()` method. This is because we know that all rectangles and circles inherit from `Shape`, and all `Shape` classes have a `getArea()` method. If a subclass inherits an abstract method from a superclass, it must provide a concrete implementation of that method, or else it will be an abstract class itself (see Figure 1.18 for a UML diagram). This approach also provides the mechanism to create other, new classes quite easily.

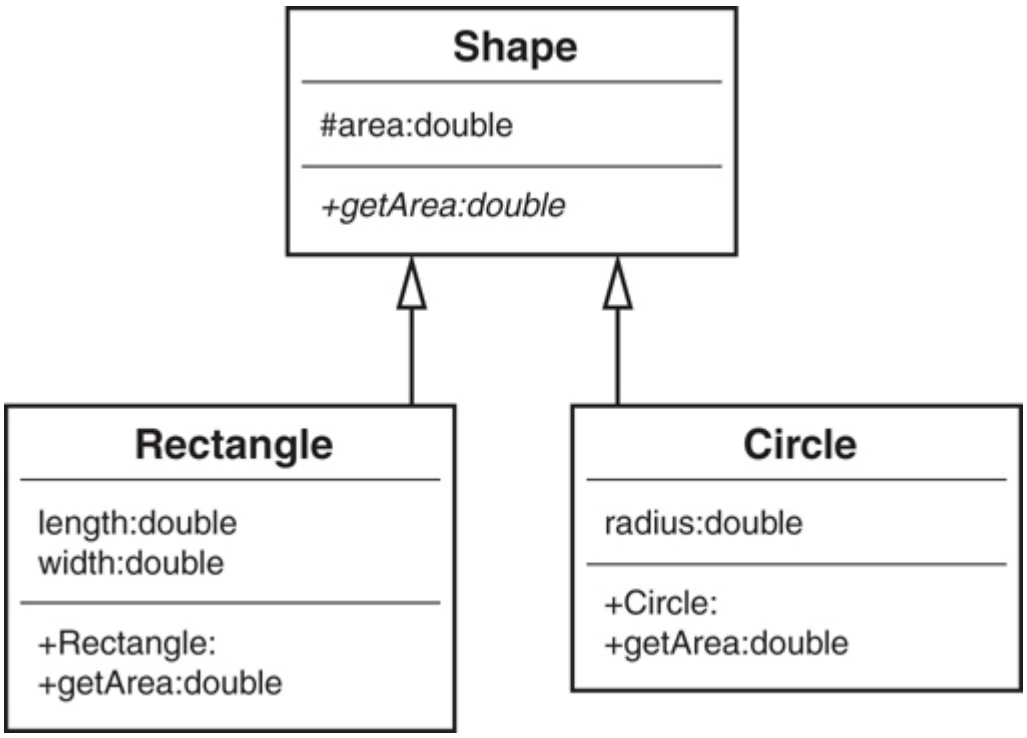

Figure 1.18 Shape UML diagram.

Thus, we can instantiate the `Shape` classes in this way:

```
Circle circle = new Circle(5);
Rectangle rectangle = new Rectangle(4,5);
```

Then, using a construct such as a stack, we can add these `Shape` classes to the stack:

```
stack.push(circle);
stack.push(rectangle);
```

> ### What Is a Stack?
>
> A stack is a data structure that is a last-in, first-out system. It is like a coin changer, where you insert coins at the top of the cylinder and, when you need a coin, you take one off the top, which is the last one you inserted. Pushing an item onto the stack means that you are adding an item to the top (like inserting another coin into the changer). Popping an item off the stack means that you are taking the last item off the stack (like taking the coin off the top).

Now comes the fun part. We can empty the stack, and we do not have to worry about what kind of `Shape` classes are in it (we just know they are shapes):

Click here to view code image

```
while ( !stack.empty()) {
    Shape shape = (Shape) stack.pop();
    System.out.println ("Area = " + shape.getArea());
}
```

In reality, we are sending the same message to all the shapes:

```
shape.getArea()
```

However, the actual behavior that takes place depends on the type of shape. For example, `Circle` calculates the area for a circle, and `Rectangle` calculates the area of a rectangle. In effect (and here is the key concept), we are sending a message to the `Shape` classes and experiencing different behavior depending on what subclass of `Shape` is being used.

This approach is meant to provide standardization for the interface across classes, as well as applications. Consider an office suite application that includes a word processing and a spreadsheet application. Let's assume that both have a class called `Office` which contains an interface called `print()`. This `print()` interface is required for all classes that are part of the office suite. The interesting thing here is that although both the word processor and the spreadsheet invoke the `print()` interface, they do different things: one prints a word processing document and the other a spreadsheet document.

> ### Polymorphism by Composition
>
> In "classical" OO, polymorphism is traditionally implemented with inheritance; however, there is a way to implement polymorphism using composition. We discuss this in Chapter 12, "The SOLID Principles of Object-Oriented Design."

Composition

It is natural to think of objects as containing other objects. A television set contains a tuner and video display. A computer contains video cards, keyboards, and drives. Although the computer can be considered an object unto itself, the drive is also considered a valid object. In fact, you could open up the computer and remove the drive and hold it in your hand. Both the computer and the drive are considered objects. It is just that the computer contains other objects—such as drives.

In this way, objects are often built, or composed, from other objects: This is composition.

Abstraction

Just as with inheritance, composition provides a mechanism for building objects. In fact, I would argue that there are only two ways to build classes from other classes: *inheritance* and *composition*. As we have seen, inheritance allows one class to inherit from another class. We can thus abstract out attributes and behaviors for common classes. For example, dogs and cats are both mammals because a dog *is-a* mammal and a cat *is-a* mammal. With composition, we can also build classes by embedding classes in other classes.

Consider the relationship between a car and an engine. The benefits of separating the engine from the car are evident. By building the engine separately, we can use the engine in various cars—not to mention other advantages. But we can't say that an engine *is-a* car. This just doesn't sound right when it rolls off the tongue (and because we are modeling real-world systems, this is the effect we want). Rather, we use the term *has-a* to describe composition relationships. A car *has-a(n)* engine.

Has-a Relationships

While inheritance is considered an *is-a* relationship, a composition relationship is termed a *has-a relationship*. Using the example in the previous section, a television *has-a* tuner and *has-a* video display. A television is obviously not a tuner, so there is no inheritance relationship. In the same vein, a computer *has*-a video card, *has-a* keyboard, and *has-a* disk drive. The topics of inheritance, composition, and how they relate to each

other are covered in great detail in Chapter 7, "Mastering Inheritance and Composition."

Conclusion

There is a lot to cover when discussing OO technologies. However, you should leave this chapter with a good understanding of the following topics:

- **Encapsulation**—Encapsulating the data and behavior into a single object is of primary importance in OO development. A single object contains both its data and behaviors and can hide what it wants from other objects.
- **Inheritance**—A class can inherit from another class and take advantage of the attributes and methods defined by the superclass.
- **Polymorphism**—Polymorphism means that similar objects can respond to the same message in different ways. For example, you might have a system with many shapes. However, a circle, a square, and a star are each drawn differently. Using polymorphism, you can send each of these shapes the same message (for example, `Draw`), and each shape is responsible for drawing itself.
- **Composition**—Composition means that an object is built from other objects.

This chapter covers the fundamental OO concepts, of which by now you should have a good grasp.

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.

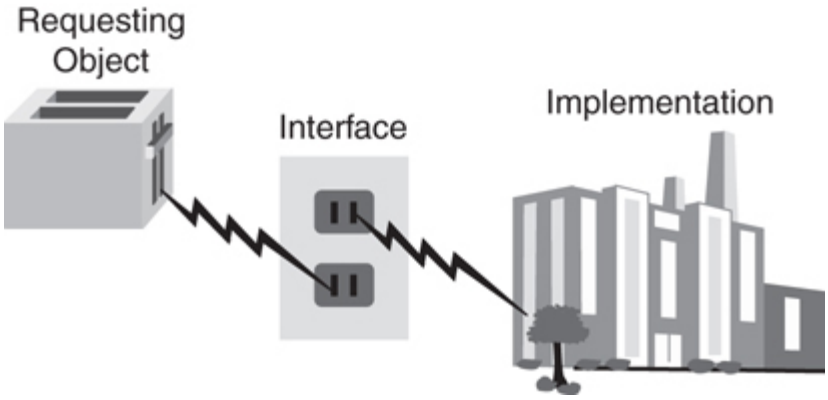

Figure 2.1 Power plant revisited.

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.

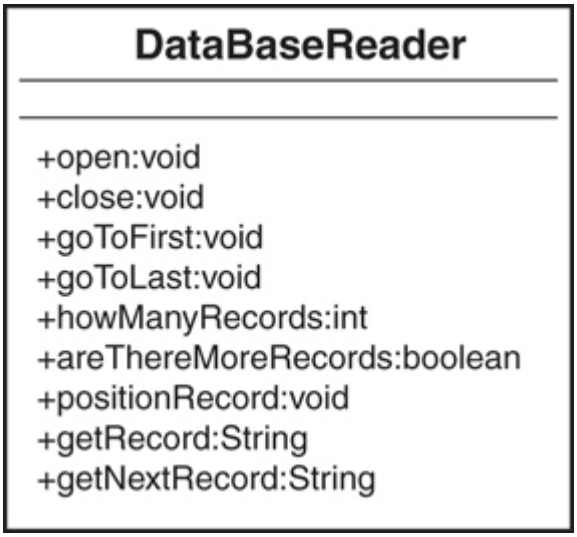

Figure 2.2 A Unified Modeling Language class diagram for the `DataBaseReader` class.

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:

Click here to view code image

```
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:

Click here to view code image

```
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.

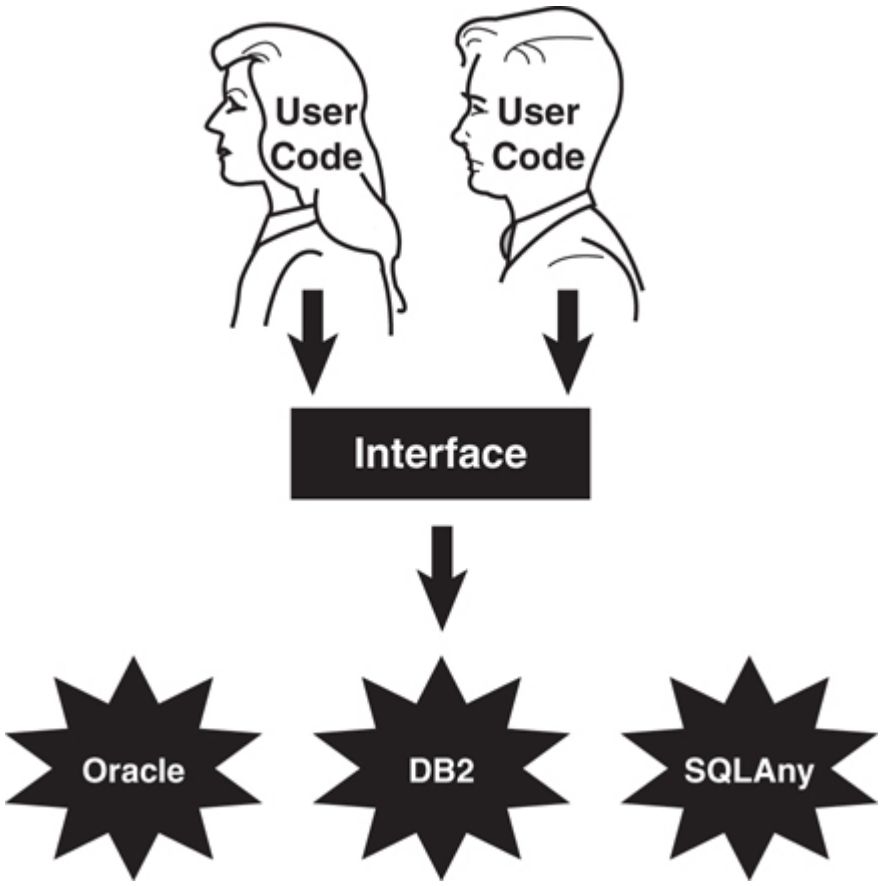

Figure 2.3 The interface.

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.

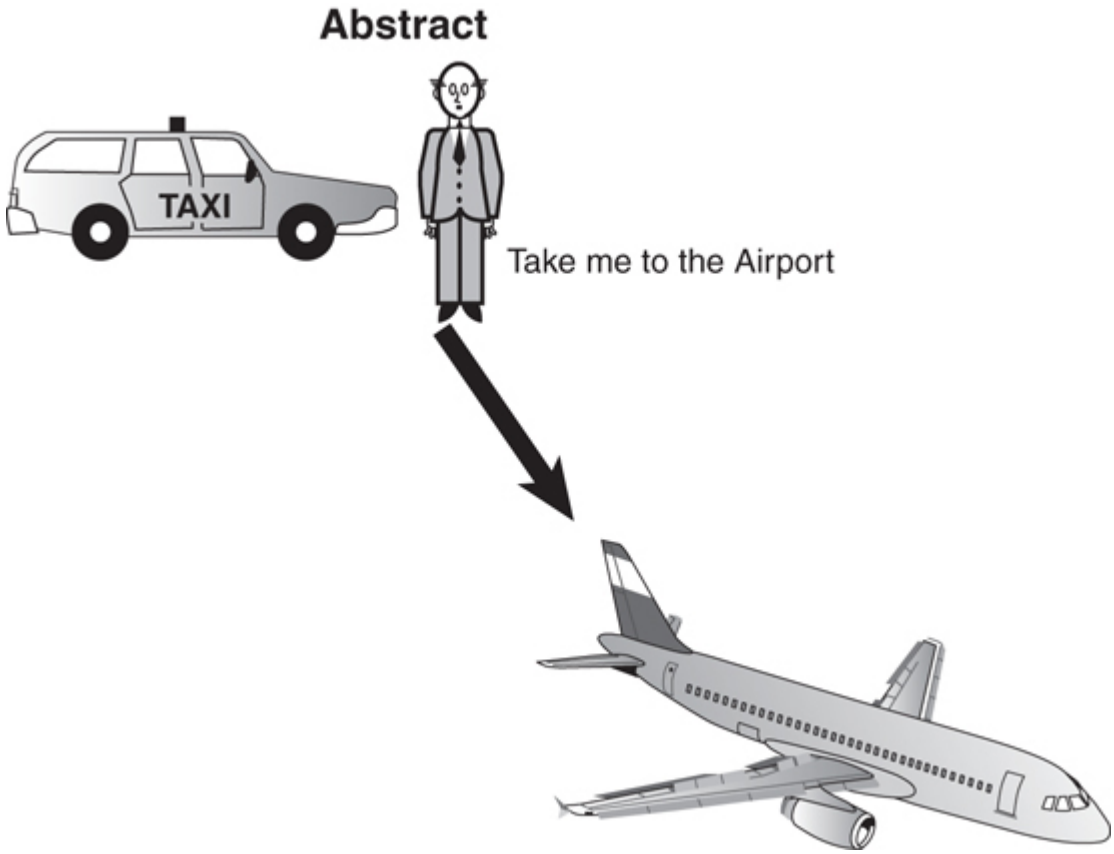

Figure 2.4 An abstract interface.

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.)

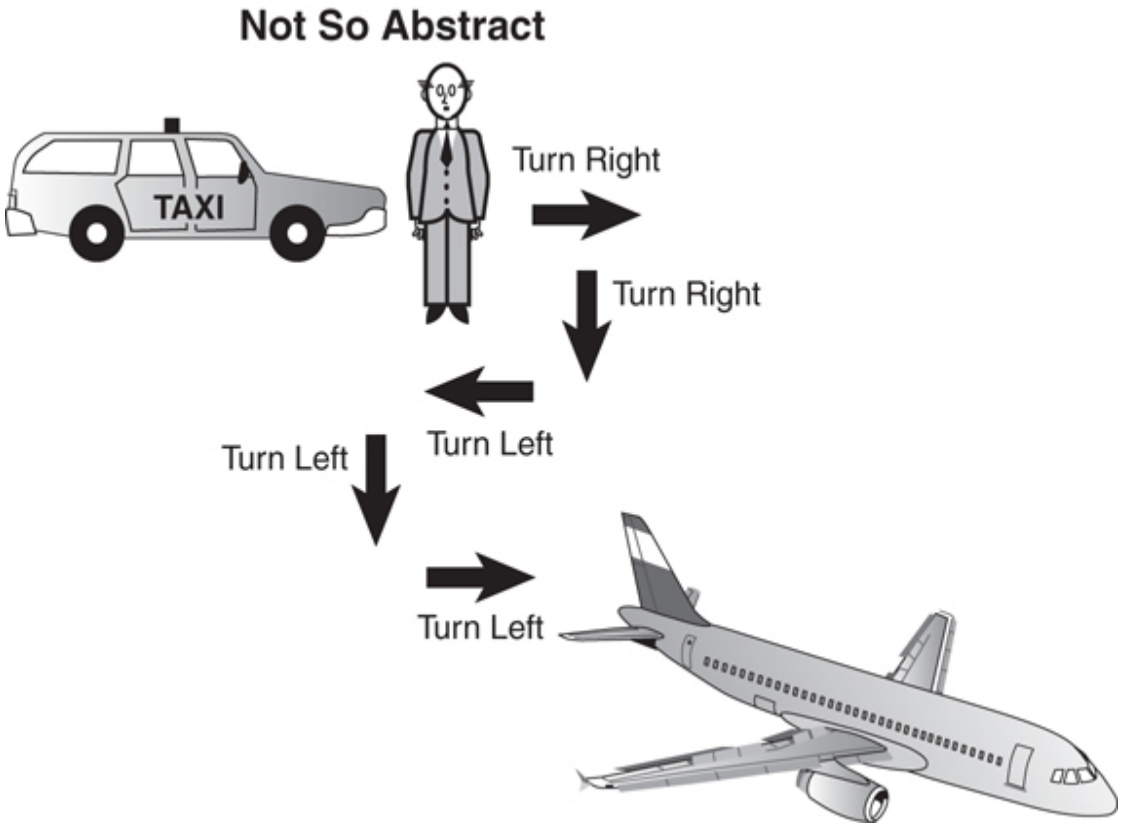

Figure 2.5 A not-so-abstract interface.

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.

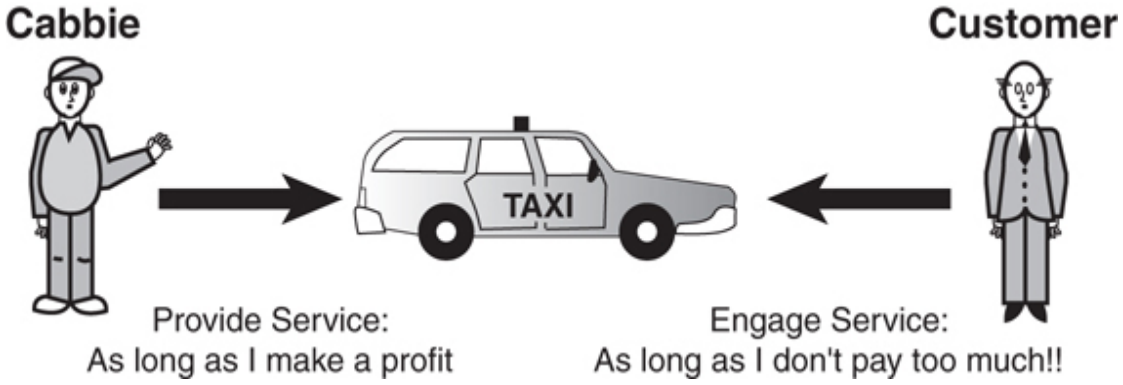

Figure 2.6 Providing services.

> **Looking Ahead**
>
> The cabbie is most likely an object as well.

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.

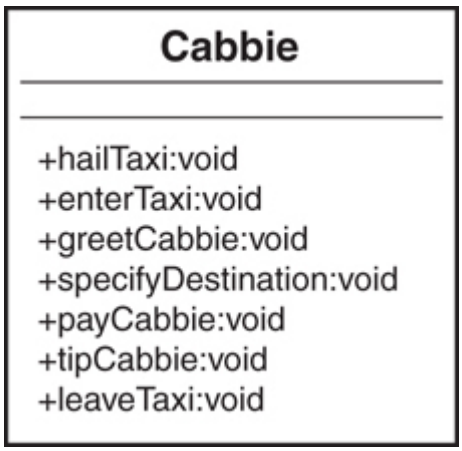

Figure 2.7 The methods in a `Cabbie` class.

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.

3. More Object-Oriented Concepts

Chapter 1, "Introduction to Object-Oriented Concepts," and Chapter 2, "How to Think in Terms of Objects," cover the basics of object-oriented (OO) concepts. Before we embark on our journey to learn some of the finer design issues relating to building an OO system, we need to cover a few more advanced OO concepts, such as constructors, operator overloading, and multiple inheritance. We also will consider error-handling techniques and how scope applies to object-oriented design.

Some of these concepts might not be vital to understanding an OO design at a higher level, but they are necessary to anyone involved in the design and implementation of an OO system.

Constructors

Constructors may be a new concept for structured programmers. Although constructors are not normally used in non-OO languages such as COBOL, C, and Basic, the *struct*, which is part of C/C++, does include constructors. In the first two chapters we alluded to these special methods that are used to *construct* objects. In some OO languages, such as Java and C#, constructors are methods that share the same name as the class. Visual Basic .NET uses the designation *New* and Swift uses the *init* keyword. As usual, we will focus on the concepts of constructors and not cover the specific syntax of all the languages. Let's take a look at some Java code that implements a constructor.

For example, a constructor for the `Cabbie` class we covered in Chapter 2 would look like this:

```
public Cabbie(){
    /* code to construct the object */
}
```

The compiler will recognize that the method name is identical to the class name and consider the method a constructor.

> **Caution**
>
> Note that in this Java code (as with C# and C++), a constructor does not have a return value. If you provide a return value, the compiler will not treat the method as a constructor.

For example, if you include the following code in the class, the compiler will not consider this a constructor because it has a return value—in this case an integer.

```
public int Cabbie(){
    /* code to construct the object */
}
```

This syntax requirement can cause problems because this code will compile but will not behave as expected.

When Is a Constructor Called?

When a new object is created, one of the first things that happens is that the constructor is called. Check out the following code:

```
Cabbie myCabbie = new Cabbie();
```

The `new` keyword creates a new instance of the `Cabbie` class, thus allocating the required memory. Then the constructor itself is called, passing the arguments in the parameter list. The constructor provides the developer the opportunity to attend to the appropriate initialization.

Thus, the code `new Cabbie()` will instantiate a `Cabbie` object and call the `Cabbie` method, which is the constructor.

What's Inside a Constructor?

Perhaps the most important function of a constructor is to initialize the memory allocated when the `new` keyword is encountered. In short, code

included inside a constructor should set the newly created object to its initial, stable, safe state.

For example, if you have a counter object with an attribute called `count`, you need to set `count` to zero in the constructor:

```
count = 0;
```

> **Initializing Attributes**
>
> In structured programming, a routine named housekeeping (or initialization) is often used for initialization purposes. Initializing attributes is a common function performed within a constructor. Regardless, don't rely on the system defaults.

The Default Constructor

If you write a class and do not include a constructor, the class will still compile, and you can still use it. If the class provides no explicit constructor, a default constructor will be provided. It is important to understand that at least one constructor always exists, regardless of whether you write a constructor yourself. If you do not provide a constructor, the system will provide a default constructor for you.

Besides the creation of the object itself, the only action that a default constructor takes is to call the constructor of its superclass. In many cases, the superclass will be part of the language framework, like the `Object` class in Java. For example, if a constructor is not provided for the `Cabbie` class, the following default constructor is inserted:

```
public Cabbie(){
    super();
}
```

If you were to decompile the bytecode produced by the compiler, you would see this code. The compiler actually inserts it.

In this case, if `Cabbie` does not explicitly inherit from another class, the `Object` class will be the parent class. Perhaps the default constructor might be sufficient in some cases; however, in most cases some sort of memory

initialization should be performed. Regardless of the situation, it is good programming practice to always include at least one constructor in a class. If there are attributes in the class, it is always good practice to initialize them. Moreover, initializing variables is always a good practice when writing code, object-oriented or not.

> ### Providing a Constructor
>
> The general rule is that you should always provide a constructor, even if you do not plan to do anything inside it. You can provide a constructor with nothing in it and then add to it later. Although there is technically nothing wrong with using the default constructor provided by the compiler, for documentation and maintenance purposes, it is always nice to know exactly what your code looks like.

It is not surprising that maintenance becomes an issue here. If you depend on the default constructor and then subsequent maintenance adds another constructor, the default constructor is no longer created. This may actually break code that was assuming the presence of a default constructor.

Always remember that the default constructor is added only if you don't include any constructors. As soon as you include just one, the default constructor is not provided.

Using Multiple Constructors

In many cases, an object can be constructed in more than one way. To accommodate this situation, you need to provide more than one constructor. For example, let's consider the `Count` class presented here:

```
public class Count {

    int count;

    public Count(){
        count = 0;
    }
}
```

On the one hand, we want to initialize the attribute `count` to count to zero: We can easily accomplish this by having a constructor initialize `count` to zero as follows:

```
public Count(){
    count = 0;
}
```

On the other hand, we might want to pass an initialization parameter that allows `count` to be set to various numbers:

```
public Count (int number){
    count = number;
}
```

This is called *overloading a method* (overloading pertains to all methods, not just constructors). Most OO languages provide functionality for overloading a method.

Overloading Methods

Overloading allows a programmer to use the same method name over and over, as long as the signature of the method is different each time. The signature consists of the method name and a parameter list (see Figure 3.1).

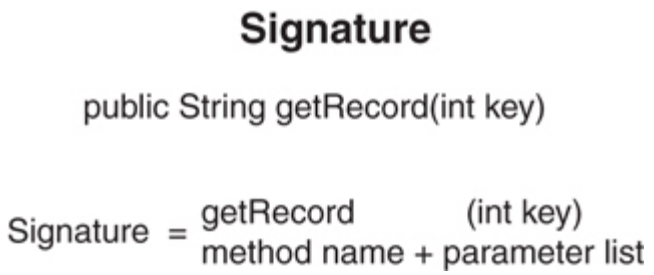

Figure 3.1 The components of a signature.

Thus, the following methods *all* have different signatures:

```
public void getCab();


// different parameter list
public void getCab (String cabbieName);

// different parameter list
public void getCab (int numberOfPassengers);
```

> ### Signatures
>
> Depending on the language, the signature may or may not include the return type. In Java and C#, the return type is not part of the signature. For example, the following methods would conflict even though the return types are different:
>
> ```
> public void getCab (String cabbieName);
> public int getCab (String cabbieName);
> ```
>
> The best way to understand signatures is to write some code and run it through the compiler.

By using different signatures, you can construct objects differently depending on the constructor used. This functionality is very helpful when you don't always know ahead of time how much information you have available. For example, when creating a shopping cart, customers may already be logged in to their account (and you will have all of their information). On the other hand, a totally new customer may be placing items in the cart with no account information available at all. In each case the constructor would initialize differently.

Using UML to Model Classes

Let's return to the database reader example we used earlier in Chapter 2. Consider that we have two ways we can construct a database reader:

- Pass the name of the database and position the cursor at the beginning of the database.
- Pass the name of the database and the position within the database where we want the cursor to position itself.

Figure 3.2 shows a class diagram for the `DataBaseReader` class. Note that the diagram lists two constructors for the class. Although the diagram shows the two constructors, without the parameter list, there is no way to know which constructor is which. To distinguish the constructors, you can look at the corresponding code in the `DataBaseReader` class listed next.

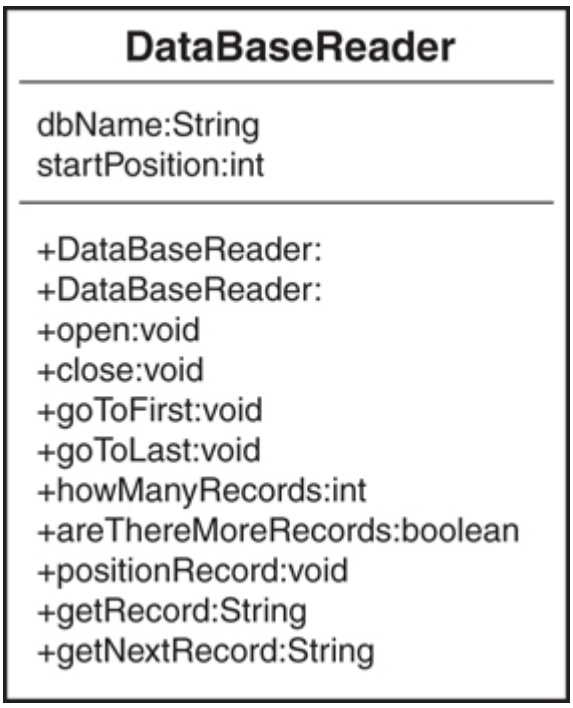

Figure 3.2 The `DataBaseReader` class diagram.

> ### No Return Type
>
> Notice that in this class diagram the constructors do not have a return type. All other methods besides constructors must have return types.

Here is a code segment of the class that shows its constructors and the attributes that the constructors initialize (see Figure 3.3):

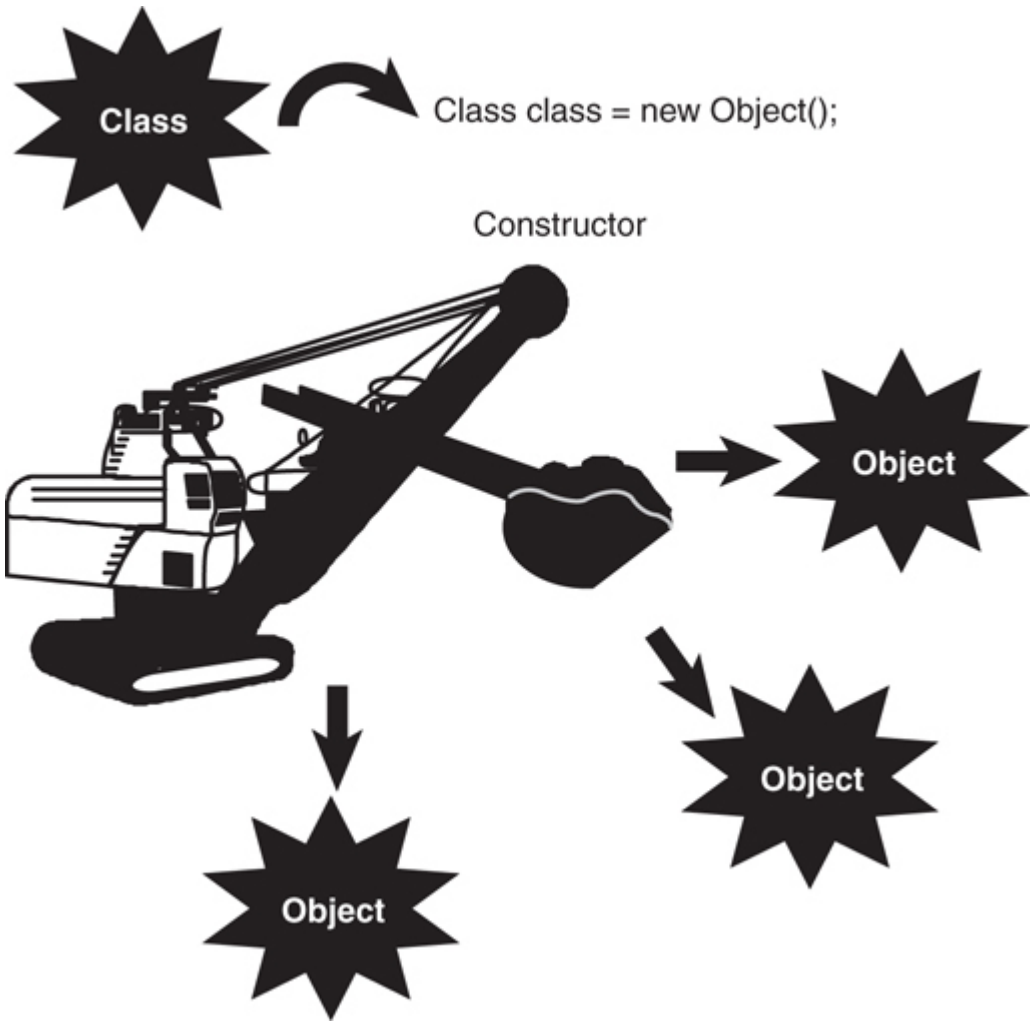

Figure 3.3 Creating a new object.

Click here to view code image

```
public class DataBaseReader {

    String dbName;
    int startPosition;

    // initialize just the name
    public DataBaseReader (String name){
        dbName = name;
        startPosition = 0;
    };

    // initialize the name and the position
    public DataBaseReader (String name, int pos){
        dbName = name;
```

```
        startPosition = pos;
    };

    .. // rest of class
}
```

Note how `startPosition` is initialized in both cases. If the constructor is not passed the information via the parameter list, it is initialized to a default value, such as `0`.

How the Superclass Is Constructed

When using inheritance, you must know how the parent class is constructed. Remember that when you use inheritance, you are inheriting everything about the parent. Thus, you must become intimately aware of all the parent's data and behavior. The inheritance of an attribute is fairly obvious; however, how a constructor is inherited is not as obvious. After the `new` keyword is encountered and the object is allocated, the following steps occur (see Figure 3.4):

1. Inside the constructor, the constructor of the class's superclass is called. If there is no explicit call to the superclass constructor, the default is called automatically; however, you can see the code in the bytecodes.

2. Each class attribute of the object is initialized. These are the attributes that are part of the class definition (instance variables), not the attributes inside the constructor or any other method (local variables). In the `DataBaseReader` code presented earlier, the integer `startPosition` is an instance variable of the class.

3. The rest of the code in the constructor executes.

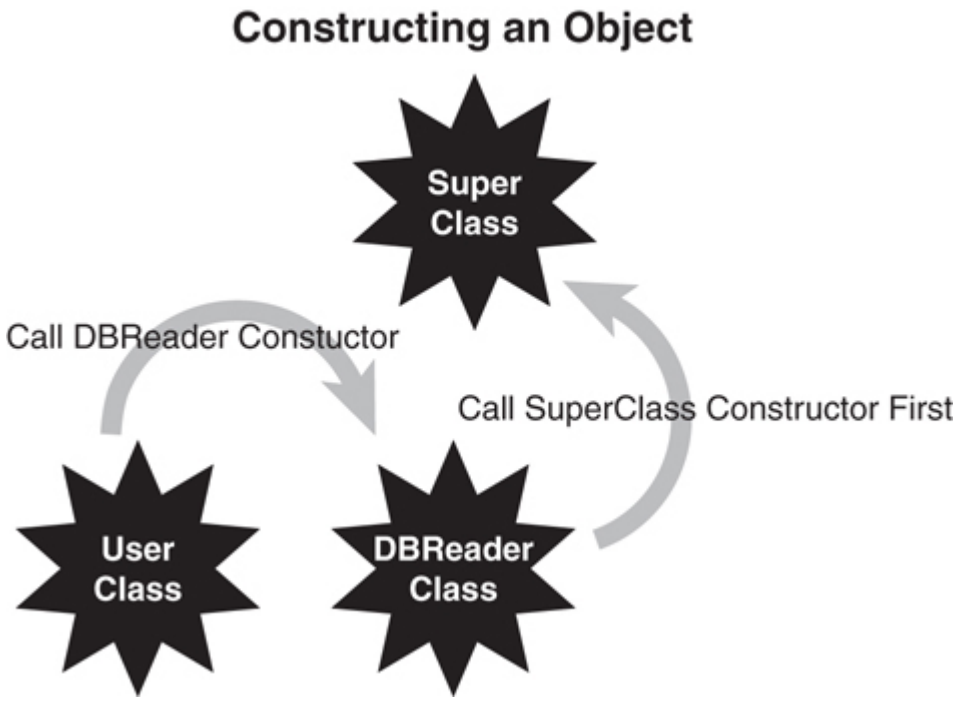

Figure 3.4 Constructing an object.

The Design of Constructors

As we have already seen, when designing a class, it is good practice to initialize all the attributes. In some languages, the compiler provides some sort of initialization. As always, don't count on the compiler to initialize attributes! In Java, you cannot use an attribute until it is initialized. If the attribute is first set in the code, make sure that you initialize the attribute to some valid condition—for example, set an integer to zero.

Constructors are used to ensure that the application is in a stable state (I like to call it a "safe" state). For example, initializing an attribute to zero, when it is intended for use as a denominator in a division operation, might lead to an unstable application. You must take into consideration that a division by zero is an illegal operation. Initializing to zero is not always the best policy.

During the design, it is good practice to identify a stable state for all attributes and then initialize them to this stable state in the constructor.

Error Handling

It is extremely rare for a class to be written perfectly the first time. In most, if not all, situations, things *will* go wrong. Any developer who does not plan for problems is inviting disaster.

Assuming that your code has the capability to detect and trap an error condition, you can handle the error in several ways: In Chapter 11 of their book *Java Primer Plus*, Tyma, Torok, and Downing (ISBN: 9781571690623) state that there are three basic solutions to handling problems that are detected in a program: fix it, ignore the problem by squelching it, or exit the runtime in some graceful manner. In Chapter 4 of their book *Object-Oriented Design in Java* (ISBN: 9781571691347), Gilbert and McCarty expand on this theme by adding the choice of throwing an exception:

- Ignore the problem—not a good idea!
- Check for potential problems and abort the program when you find a problem.
- Check for potential problems, catch the mistake, and attempt to fix the problem.
- Throw an exception. (Often this is the preferred way to handle the situation.)

These strategies are discussed in the following sections.

Ignoring the Problem

Simply ignoring a potential problem is a recipe for disaster. And if you are going to ignore the problem, why bother detecting it in the first place? It is obvious that you should not ignore any known problem. The primary directive for all applications is that the application should never crash. If you do not handle your errors, the application will eventually terminate ungracefully or continue in a mode that can be considered an unstable state—possibly with corrupted data. In the latter case, you might not even know you are getting incorrect results, and that can be much worse than a program crash.

Checking for Problems and Aborting the Application

If you choose to check for potential problems and abort the application when a problem is detected, the application can display a message indicating that a problem exists. In this case the application gracefully exits, and the user is left staring at the computer screen, shaking her head and wondering what just happened. Although this is a far superior option to ignoring the problem, it is by no means optimal. However, this does allow the system to clean up things and put itself in a more stable state, such as closing files and forcing a system restart.

Checking for Problems and Attempting to Recover

Checking for potential problems, catching the mistake, and attempting to recover is a far superior solution than simply checking for problems and aborting. In this case, the problem is detected by the code, and the application attempts to fix itself. This works well in certain situations. For example, consider the following code:

```
if (a == 0)
    a=1;

c = b/a;
```

It is obvious that if the conditional statement is not included in the code, and a zero makes its way to the divide statement, you will get a system exception because you cannot divide by zero. By catching the exception and setting the variable `a` to `1`, at least the system will not crash. However, setting `a` to `1` might not be a proper solution because the result would be incorrect. The better solution would be to prompt the user to reenter the proper input value.

> ### A Mix of Error-Handling Techniques
>
> Despite the fact that this type of error handling is not necessarily object-oriented in nature, I believe that it has a valid place in OO design. Throwing an exception (discussed inthe next section) can be expensive in terms of overhead. Thus, although exceptions may be a valid design choice, you will still want to consider other error-

> handling techniques (even tried-and-true structured techniques), depending on your design and performance needs.

Although the error checking techniques mentioned previously are preferable to doing nothing, they still have a few problems. It is not always easy to determine where a problem first appears. And it might take a while for the problem to be detected. In any event, it is beyond the scope of this book to explain error handling in great detail. However, it is important to design error handling into the class right from the start, and often the operating system itself can alert you to problems that it detects.

Throwing an Exception

Most OO languages provide a feature called *exceptions*. In the most basic sense, exceptions are unexpected events that occur within a system. Exceptions provide a way to detect problems and then handle them. In Java, C#, C++, Swift, and Visual Basic, exceptions are handled by the keywords `catch` and `throw`. This might sound like a baseball game, but the key concept here is that a specific block of code is written to handle a specific exception. This solves the problem of trying to figure out where the problem started and unwinding the code to the proper point.

Here is the structure for a Java `try/catch` block:

```
try {

    // possible nasty code

} catch(Exception e) {

    // code to handle the exception
}
```

If an exception is thrown within the `try` block, the `catch` block will handle it. When an exception is thrown while the block is executing, the following occurs:

1. The execution of the `try` block is terminated.

2. The `catch` clauses are checked to determine whether an appropriate `catch` block for the offending exception was included. (There might be

more than one `catch` clause per `try` block.)

3. If none of the `catch` clauses handles the offending exception, it is passed to the next higher-level `try` block. (If the exception is not caught in the code, the system ultimately catches it, and the results are unpredictable—that is, an application crash.)

4. If a `catch` clause is matched (the first match encountered), the statements in the `catch` clause are executed.

5. Execution then resumes with the statement following the `try` block.

Suffice it to say that exceptions are an important advantage for OO programming languages. Here is an example of how an exception is caught in Java:

Click here to view code image

```
try {

    // possible nasty code
    count = 0;
    count = 5/count;

} catch(ArithmeticException e) {

    // code to handle the exception
    System.out.println(e.getMessage());
    count = 1;

}
System.out.println("The exception is handled.");
```

> ### Exception Granularity
>
> You can catch exceptions at various levels of granularity. You can catch all exceptions or check for specific exceptions, such as arithmetic exceptions. If your code does not catch an exception, the Java runtime will—and it won't be happy about it!

In this example, the division by zero (because `count` is equal to `0`) within the `try` block will cause an arithmetic exception. If the exception was

generated (thrown) outside a `try` block, the program would most likely have been terminated (crashed). However, because the exception was thrown within a `try` block, the `catch` block is checked to see whether the specific exception (in this case, an arithmetic exception) was planned for. Because the `catch` block contains a check for the arithmetic exception, the code within the `catch` block is executed, thus setting `count` to `1`. After the `catch` block executes, the `try/catch` block is exited, and the message `The exception is handled.` appears on the Java console. The logical flow of this process is illustrated in Figure 3.5.

Figure 3.5 Catching an exception.

If you had not put `ArithmeticException` in the `catch` block, the program would likely have crashed. You can catch all exceptions by using the following code:

```
try {

    // possible nasty code

} catch(Exception e) {

    // code to handle the exception
}
```

The `Exception` parameter in the `catch` block is used to catch any exception that might be generated within the scope of a `try` block.

> ### Bulletproof Code
>
> It's a good idea to use a combination of the methods described here to make your program as bulletproof to your user as possible.

The Importance of Scope

Multiple objects can be instantiated from a single class. Each of these objects has a unique identity and state. This is an important point. Each object is constructed separately and is allocated its own separate memory. However, some attributes and methods may, if properly declared, be shared by all the objects instantiated from the same class, thus sharing the memory allocated for these class attributes and methods.

> ### A Shared Method
>
> A constructor is a good example of a method that is shared by all instances of aclass.

Methods represent the behaviors of an object; the state of the object is represented by attributes. There are three types of attributes:

- Local attributes
- Object attributes
- Class attributes

Local Attributes

Local attributes are owned by a specific method. Consider the following code:

```
public class Number {

    public method1() {
        int count;

    }

    public method2() {

    }

}
```

The method `method1` contains a local variable called `count`. This integer is accessible only inside `method1`. The method `method2` has no idea that the integer `count` even exists.

At this point, we introduce a very important concept: scope. Attributes (and methods) exist within a particular scope. In this case, the integer count exists within the scope of `method1`. In Java, C#, C++, and Swift, scope is delineated by curly braces (`{}`). In the `Number` class, there are several possible scopes—just start matching the curly braces.

The class itself has its own scope. Each instance of the class (that is, each object) has its own scope. Both `method1` and `method2` have their own scopes as well. Because `count` lives within `method1`'s curly braces, when `method1` is invoked, a copy of `count` is created. When `method1` terminates, the copy of `count` is removed.

For some more fun, look at this code:

```
public class Number {

    public method1() {
        int count;
    }

    public method2() {
        int count;
    }

}
```

This example has two copies of an integer `count` in this class. Remember that `method1` and `method2` each has its own scope. Thus, the compiler can tell which copy of `count` to access simply by recognizing which method it is in. You can think of it in these terms:

```
method1.count;

method2.count;
```

As far as the compiler is concerned, the two attributes are easily differentiated, even though they have the same name. It is almost like two

people having the same last name, but based on the context of their first names, you know that they are two separate individuals.

Object Attributes

In many design situations, an attribute must be shared by several methods within the same object. In Figure 3.6, for example, three objects have been constructed from a single class. Consider the following code:

Click here to view code image

```
public class Number {

    int count;    // available to both method1 and method2

    public method1() {
        count = 1;
    }

    public method2() {
        count = 2;
    }

}
```

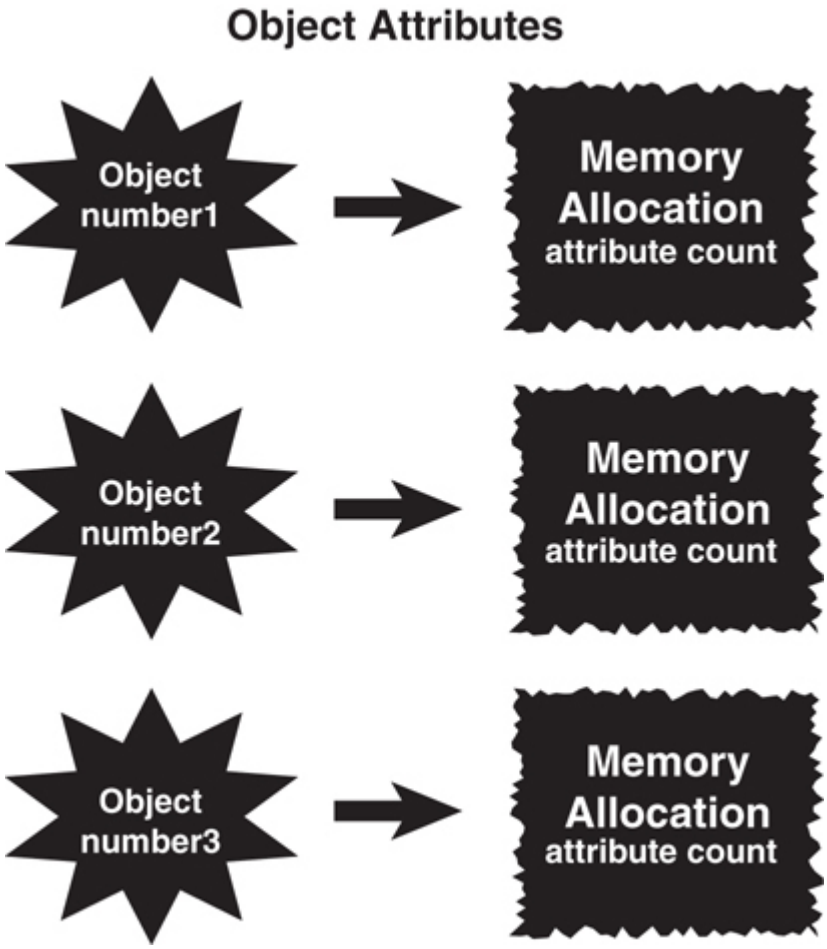

Figure 3.6 Object attributes.

Note here that the class attribute `count` is declared outside the scope of both `method1` and `method2`. However, it is within the scope of the class. Thus, `count` is available to both `method1` and `method2`. (Basically, all methods in the class have access to this attribute.) Notice that the code for both methods is setting `count` to a specific value. There is only one copy of `count` for the entire object, so both assignments operate on the same copy in memory. However, this copy of `count` is not shared between different objects.

To illustrate, let's create three copies of the `Number` class:

```
Number number1 = new Number();
Number number2 = new Number();
Number number3 = new Number();
```

Each of these objects—`number1`, `number2`, and `number3`—is constructed separately and is allocated its own resources. There are three separate instances of the integer `count`. When `number1` changes its attribute `count`, this in no way affects the copy of `count` in object `number2` or object `number3`. In this case, integer `count` is an *object attribute*.

You can play some interesting games with scope. Consider the following code:

Click here to view code image

```
public class Number {

    int count;

    public method1() {
        int count;
    }

    public method2() {
        int count;
    }

}
```

In this case, three totally separate memory locations have the name of `count` for each object. The object owns one copy, and `method1()` and `method2()` each have their own copy.

To access the object variable from within one of the methods, say `method1()`, you can use a pointer called `this` in the C-based languages:

```
public method1() {
        int count;

        this.count = 1;
}
```

Notice that some code looks a bit curious:

```
this.count = 1;
```

The selection of the word `this` as a keyword is perhaps unfortunate. However, we must live with it. The use of the `this` keyword directs the compiler to access the object variable `count` and not the local variables within the method bodies.

> **Note**
>
> The keyword this is a reference to the current object.

Class Attributes

As mentioned earlier, it is possible for two or more objects of the same class to share attributes. In Java, C#, C++, and Swift, you do this by making the attribute *static*:

```
public class Number {

    static int count;

    public method1() {
    }

}
```

By declaring `count` as static, this attribute is allocated a single piece of memory for all objects instantiated from the class. Thus, all objects of the class use the same memory location for `count`. Essentially, each class has a single copy, which is shared by all objects of that class (see Figure 3.7). This is about as close to global data as we get in OO design.

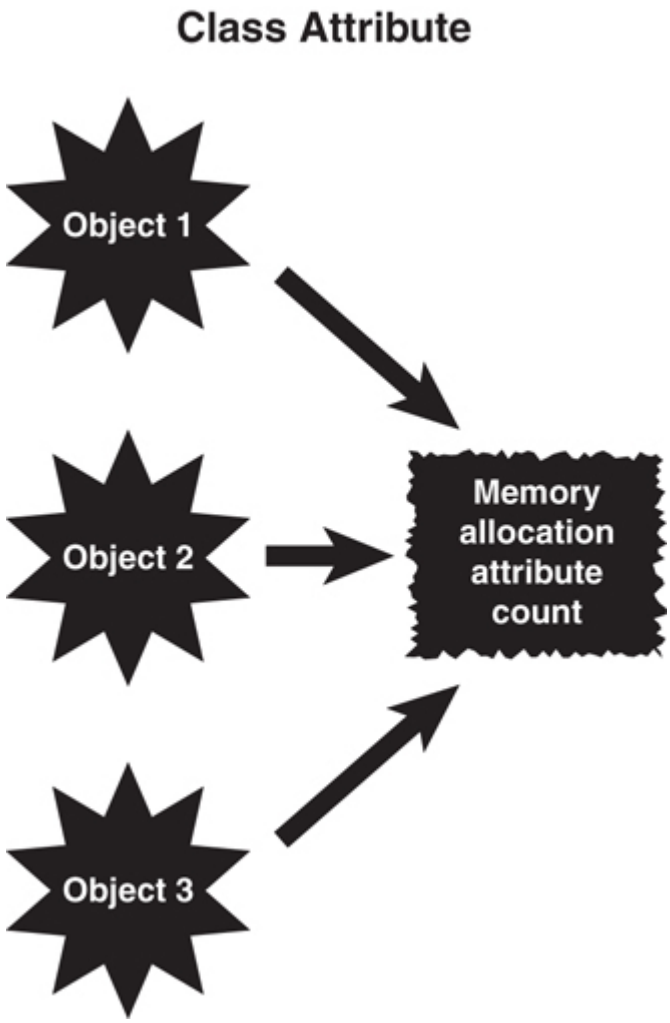

Figure 3.7 Class attributes.

There are many valid uses for class attributes; however, you must be aware of potential synchronization problems. Let's instantiate two `Count` objects:

```
Count Count1 = new Count();
Count Count2 = new Count();
```

For the sake of argument, let's say that the object `Count1` is going merrily about its way and is using `count` as a means of keeping track of the pixels on a computer screen. This is not a problem until the object `Count2` decides to use attribute `count` to keep track of sheep. The instant that `Count2` records its first sheep, the data that `Count1` was saving is lost. In practice, there might not be a lot of uses for static methods. Make sure you are confident in their use before incorporating them in designs.

Operator Overloading

Some OO languages enable you to overload an operator. C++ is an example of one such language. Operator overloading enables you to change the meaning of an operator. For example, when most people see a plus sign, they assume it represents addition. If you see the equation

```
X = 5 + 6;
```

you expect that `X` would contain the value `11`. And in this case, you would be correct.

However, at times a plus sign could represent something else. For example, in the following code:

```
String firstName = "Joe", lastName = "Smith";

String Name = firstName + " " + lastName;
```

You would expect that `Name` would contain `Joe Smith`. The plus sign here has been overloaded to perform string concatenation.

> ### String Concatenation
>
> *String concatenation* occurs when two separate strings are combined to create a new, single string.

In the context of strings, the plus sign does not mean addition of integers or floats but concatenation of strings.

What about matrix addition? You could have code like this:

```
Matrix a, b, c;

c = a + b;
```

Thus, the plus sign now performs matrix addition, not addition of integers or floats.

Overloading is a powerful mechanism. However, it can be downright confusing for people who read and maintain code. In fact, developers can confuse themselves. To take this to an extreme, it would be possible to

change the operation of addition to perform subtraction. Why not? Operator overloading allows you to change the meaning of an operator. Thus, if the plus sign were changed to perform subtraction, the following code would result in an `X` value of `-1`.

```
x = 5 + 6;
```

Although some languages may not allow programmers to overload operators directly, many languages do implement operator overloading as a feature (a common example is the plus sign for concatenation). When utilizing operator overloading, take care to document your logic so that people maintaining the code (perhaps even a future you) understand what is happening.

Multiple Inheritance

We cover inheritance in much more detail in Chapter 7, "Mastering Inheritance and Composition." However, this is a good place to begin discussing multiple inheritance, which is one of the more powerful and challenging aspects of class design.

As the name implies, *multiple inheritance* allows a class to inherit from more than one class. In practice, this seems like a great idea. Objects are supposed to model the real world, are they not? And many real-world examples of multiple inheritance exist. Parents are a good example of multiple inheritance. Each child has two parents—that's just the way it is. So it makes sense that you can design classes by using multiple inheritance. In some OO languages, such as C++, you can.

However, this situation falls into a category similar to operator overloading. Multiple inheritance is a very powerful technique, and in fact, some problems are quite difficult to solve without it. Multiple inheritance can even solve some problems quite elegantly. However, multiple inheritance can significantly increase the complexity of a system, both for the programmer and the compiler writers.

The designers of Java, .NET, and Swift decided that the increased complexity of allowing multiple inheritance far outweighed its advantages, so they simply did not implement it. In some ways, Java, .NET, and Swift

compensate for this; however, the bottom line is that Java, .NET, and Swift do not allow conventional multiple inheritance.

The modern concept of inheritance is that you can only inherit attributes from a single parent (single inheritance). Even though you can use multiple interfaces or protocols, this is not truly multiple inheritance.

> ### Behavioral and Implementation Inheritance
>
> Interfaces are a mechanism for behavioral inheritance, whereas abstract classes are used for implementation inheritance. The bottom line is that interface language constructs provide behavioral interfaces but no implementation, whereas abstract classes may provide both interfaces and implementation. This topic is covered in great detail in Chapter 8, "Frameworks and Reuse: Designing with Interfaces and Abstract Classes."

Object Operations

Some of the most basic operations in programming become more complicated when you're dealing with complex data structures and objects. For example, when you want to copy or compare primitive data types, the process is quite straightforward. However, copying and comparing objects is not quite as simple. In his book *Effective C++*, Scott Meyers devotes an entire section to copying and assigning objects.

> ### Classes and References
>
> The problem with complex data structures and objects is that they might contain references. Simply making a copy of the reference does not copy the data structures or the object that it references. In the same vein, when comparing objects, simply comparing a pointer to another pointer only compares the references—not what they point to.

The problems arise when comparisons and copies are performed on objects. Specifically, the question boils down to whether you follow the pointers. Regardless, there should be a way to copy an object. Again, this is not as

simple as it might seem. Because objects can contain references, these reference trees must be followed to do a valid copy (if you truly want to do a deep copy).

> ### Deep Versus Shallow Copies
>
> A *deep copy* occurs when all the references are followed and new copies are created for all referenced objects. Many levels might be involved in a deep copy. For objects with references to many objects, which in turn might have references to even more objects, the copy itself can create significant overhead. A shallow copy would simply copy the reference and not follow the levels. Gilbert and McCarty have a good discussion about what shallow and deep hierarchies are in *Object-Oriented Design in Java* in a section called "Prefer a Tree to a Forest."

To illustrate, in Figure 3.8, if you do a simple copy of the object (called a *bitwise copy*), only the references are copied—not any of the actual objects. Thus, both objects (the original and the copy) will reference (point to) the same objects. To perform a complete copy, in which all reference objects are copied, you must write code to create all the subobjects.

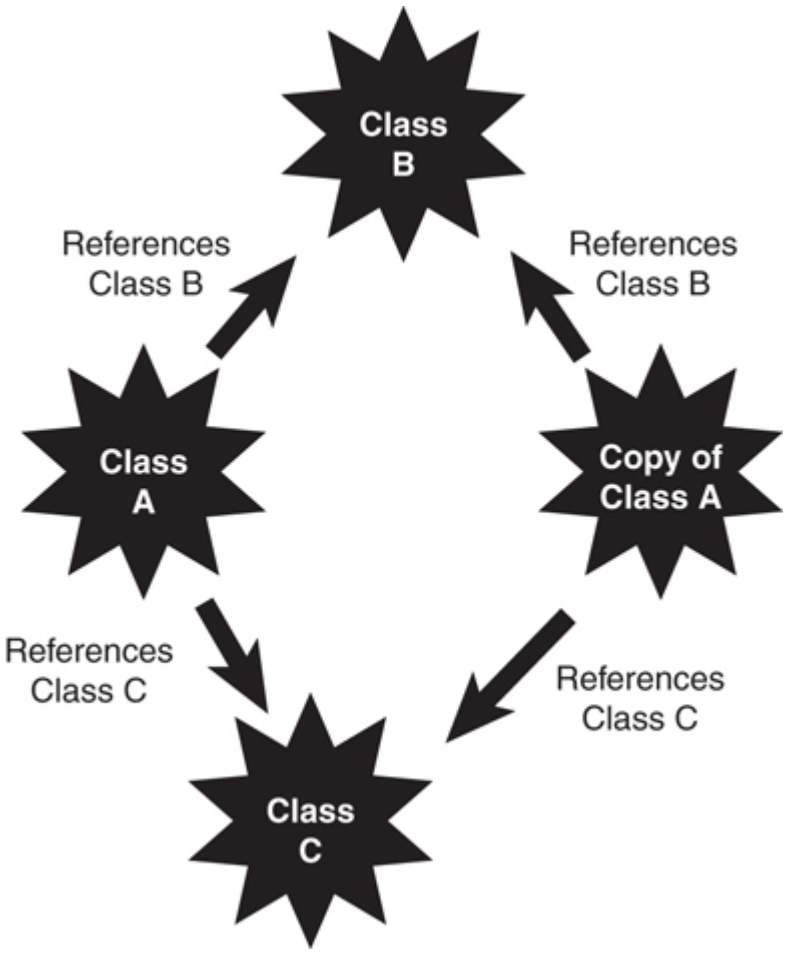

Figure 3.8 Following object references.

This problem also manifests itself when comparing objects. As with the copy function, this is not as simple as it might seem. Because objects contain references, these reference trees must be followed to do a valid comparison of objects. In most cases, languages provide a default mechanism to compare objects. As is usually the case, do not count on the default mechanism. When designing a class, you should consider providing a comparison function in your class that you know will behave as you want it to.

Conclusion

This chapter covered a number of advanced OO concepts that, although perhaps not vital to a general understanding of OO concepts, are quite necessary in higher-level OO tasks, such as designing a class. In Chapter 4,

"The Anatomy of a Class," we start looking specifically at how to design and build a class.

References

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

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

Tyma, Paul, Gabriel Torok, and Troy Downing. 1996. *Java Primer Plus*. Berkeley, CA: The Waite Group.

4. The Anatomy of a Class

In previous chapters we have covered the fundamental object-oriented (OO) concepts and determined the difference between the interface and the implementation. No matter how well you think out the problem of what should be part of the interface and what should be part of the implementation, the bottom line always comes down to how useful the class is and how it interacts with other classes. A class should never be designed in a vacuum, for as might be said, no class is an island. When objects are instantiated, they almost always interact with other objects. An object can also be part of another object or be part of an inheritance hierarchy.

This chapter examines a simple class and then takes it apart piece by piece along with guidelines that you should consider when designing classes. We will continue using the cabbie example presented in Chapter 2, "How to Think in Terms of Objects."

Each of the following sections covers a particular aspect of a class. Although not all components are necessary in every class, it is important to understand how a class is designed and constructed.

> **note**
>
> This example class is meant for illustration purposes only. Some of the methods are not fleshed out (meaning that there is no implementation) and simply present the interface—primarily to emphasize that the interface is the focus of the initial design.

The Name of the Class

The name of the class is important for several reasons. The obvious reason is to identify the class itself. Beyond simple identification, the name must be descriptive. The choice of a name is important because it provides

information about what the class does and how it interacts within larger systems.

The name is also important when considering language constraints. For example, in Java, the public class name must be the same as the filename. If these names do not match, the application won't compile.

Figure 4.1 shows the class that will be examined. Plain and simple, the name of the class in our example, `Cabbie`, is the name located after the keyword `class`:

```
public class Cabbie {

}
```

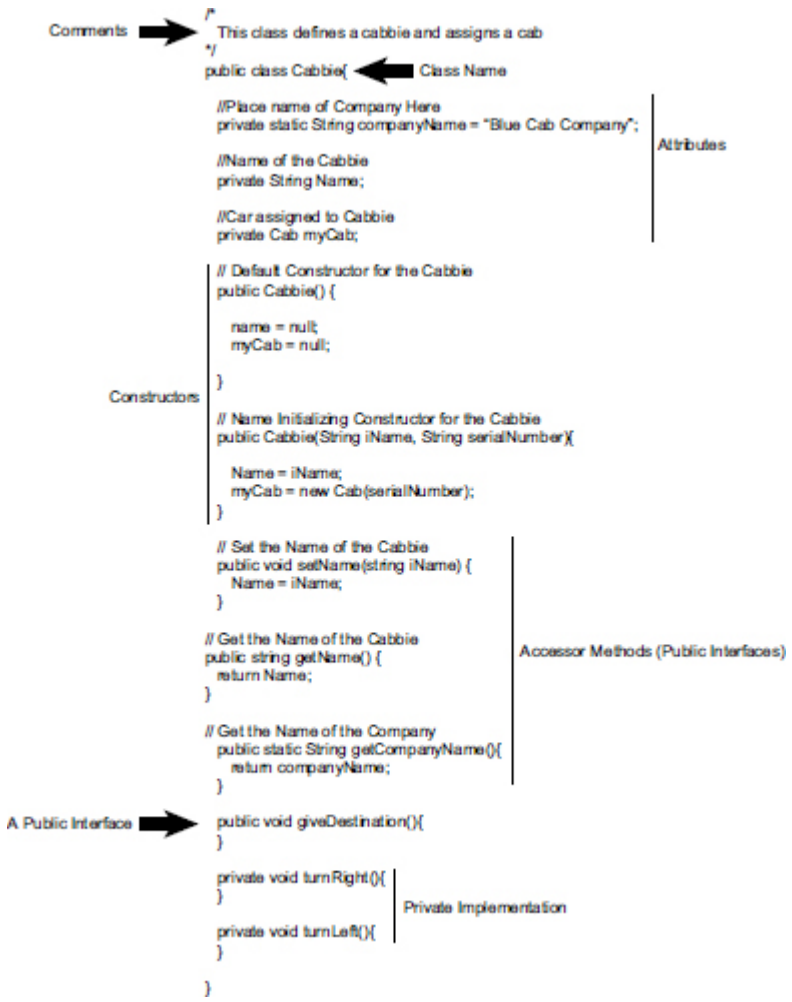

Figure 4.1 Our sample class.

> ### Using Java Syntax
>
> Remember that the convention for this book is to use Java syntax. The syntax will be similar but, perhaps, somewhat different in other languages.

The class `Cabbie` name is used whenever this class is instantiated.

Comments

Regardless of the syntax of the comments used, they are vital to understanding the function of a class. In Java and other languages, two kinds of comments are common.

> ### The Extra Java and C# Comment Style
>
> In Java and C#, there are three types of comments. In Java, the third comment type (/** */) relates to a form of documentation that Java provides. We do not cover this type of comment in this book. In C#, the syntax to create documentation comments is ///, much like the /** */ Javadoc documentation comments.

The first comment is the old C-style comment, which uses `/*` (slash-asterisk) to open the comment and `*/` (asterisk-slash) to close the comment. This type of comment can span more than one line, and it's important not to forget to use the pair of open and close comment symbols for each comment. If you miss the closing comment (`*/`), some of your code might be tagged as a comment and ignored by the compiler. Here is an example of this type of comment used with the `Cabbie` class:

Click here to view code image

```
/*

  This class defines a cabbie and assigns a cab

*/
```

The second type of comment is the `//` (slash-slash), which renders everything after it, to the end of the line, a comment. This type of comment

spans only one line, so you don't need to remember to use a close comment symbol, but you do need to remember to confine the comment to just one line and not include any live code after the comment. Here is an example of this type of comment used with the `Cabbie` class:

```
// Name of the cabbie
```

Attributes

Attributes represent the state of the object because they store the information about the object. For our example, the `Cabbie` class has attributes that store the name of the company, the name of the cabbie, and the cab assigned to the cabbie. For example, the first attribute stores the name of the company:

Click here to view code image

```
private static String companyName = "Blue Cab Company";
```

Note here the two keywords `private` and `static`. The keyword `private` signifies that a method or variable can be accessed only within the declaring object.

> ### Hiding as Much Data as Possible
>
> All the attributes in this example are private. This is in keeping with the design principle of keeping the interface design as minimal as possible. The only way to access these attributes is through the method interfaces provided (which we explore later in this chapter).

The `static` keyword signifies that there will be only one copy of this attribute for all the objects instantiated by this class. Basically, this is a class attribute. (See Chapter 3, "More Object-Oriented Concepts," for further discussion on class attributes.) Thus, even if 500 objects are instantiated from the `Cabbie` class, only one copy will be in memory of the `companyName` attribute (see Figure 4.2).

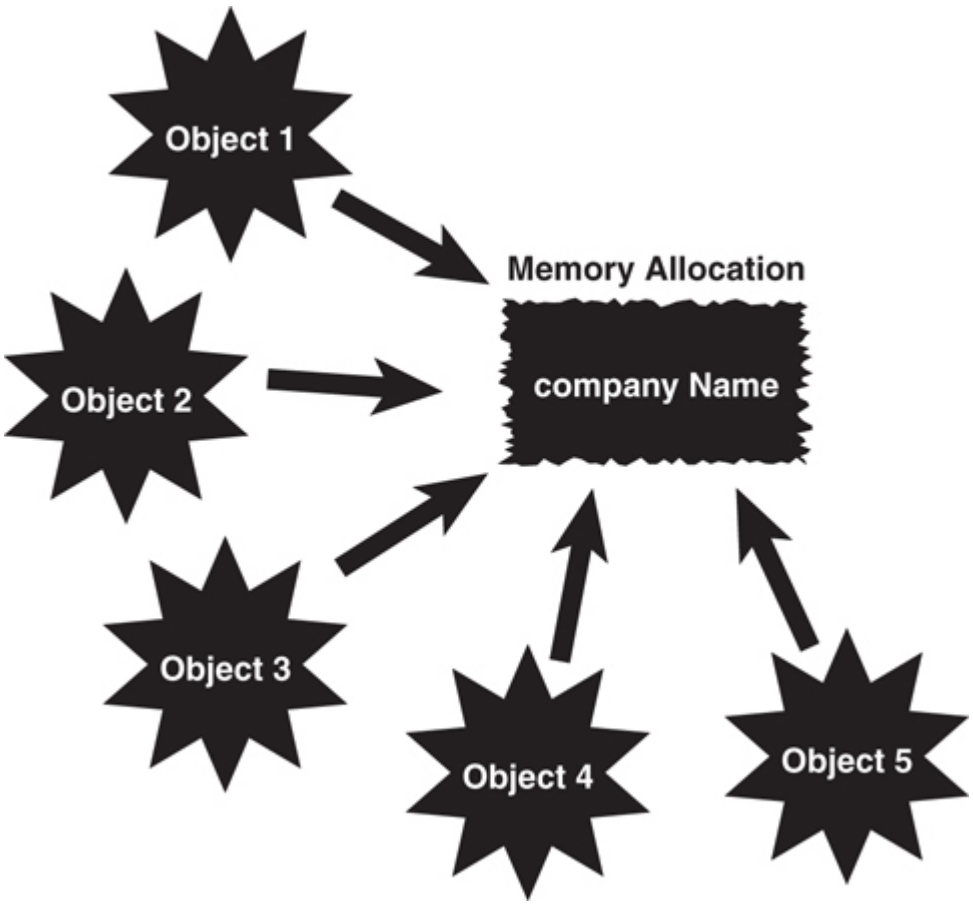

Figure 4.2 Object memory allocation.

The second attribute, `name`, is a string that stores the name of the cabbie:

```
private String name;
```

This attribute is also private so that other objects cannot access it directly. They must use the interface methods.

The `myCab` attribute is a reference to another object. The class, called `Cab`, holds information about the cab, such as its serial number and maintenance records:

```
private Cab myCab;
```

> ### Passing a Reference
>
> It is likely that the Cab object was created by another object. Thus, the object reference would be passed to the `Cabbie` object. However, for the sake of this example, the Cab is created within the

> `Cabbie` object. As a result, we are not really interested in the internals of the Cab object.

Note that at this point, only a reference to a `Cab` object is created; there is no memory allocated by this definition.

Constructors

This `Cabbie` class contains two constructors. We know they are constructors because they have the same name as the class: `Cabbie`. The first constructor is the default constructor:

```
public Cabbie() {

    name = null;
    myCab = null;

}
```

Technically, this is not a default constructor provided by the system. Recall that the compiler will provide an empty default constructor if you do not code any constructor for a class. By definition, the reason it is called a default constructor here is because it is a constructor with no arguments, which, in effect, overrides the compiler's default constructor.

If you provide a constructor with arguments, the system will not provide a default constructor. Although this can seem complicated, the rule is that the compiler's default constructor is included only if you provide *no* constructors in your code.

> ### No Constructor
>
> Coding no constructor and allowing the default constructor to be in play can cause maintenance headaches down the road. If the code relies on a default constructor, and another constructor is added later, the default constructor will not be included by the system.

In this constructor, the attributes `name` and `myCab` are set to `null`:

```
name = null;
myCab = null;
```

The Nothingness of `Null`

In many programming languages, the value `null` represents a value of nothing. This might seem like an esoteric concept, but setting an attribute to nothing is a useful programming tech-nique. Checking a variable for `null` can identify whether a value has been properly initialized. For example, you might want to declare an attribute that will later require user input. Thus, you can initialize the attribute to null before the user is given the opportunity to enter the data. By setting the attribute to `null` (which is a valid condition), you can check whether an attribute has been properly set. Note that in some languages this is not allowed with the string type. In .NET for example, it is required to use name = `string.empty;`.

As we know, it is always a good coding practice to initialize attributes in the constructors. In the same vein, it's a good programming practice to then test the value of an attribute to see whether it is `null`. This can save you a lot of headaches later if the attribute or object was not set properly. For example, if you use the `myCab` reference before a real object is assigned to it, you will most likely have a problem. If you set the `myCab` reference to `null` in the constructor, you can later check to see whether `myCab` is still `null` when you attempt to use it. An exception might be generated if you treat an uninitialized reference as if it were properly initialized.

Consider another example: If you have an Employee class that includes a spouse attribute (perhaps for insurance purposes), you'd better make provisions for the situation when an employee is not married. By initially setting the attribute to `null`, you can then check for this status.

The second constructor provides a way for the user of the class to initialize the `Name` and `myCab` attributes:

Click here to view code image

```
public Cabbie(String iName, String serialNumber) {

    name = iName;
    myCab = new Cab(serialNumber);

}
```

In this case, the user would provide two strings in the parameter list of the constructor to properly initialize attributes. Notice that the `myCab` object is instantiated in this constructor:

```
myCab = new Cab(serialNumber);
```

As a result of executing this line of code, the storage for a `Cab` object is allocated. Figure 4.3 illustrates how a new instance of a `Cab` object is referenced by the attribute `myCab`. Using two constructors in this example demonstrates a common use of method overloading. Notice that the constructors are all defined as `public`. This makes sense because in this case, the constructors are obvious members of the class interface. If the constructors were private, other objects couldn't access them—thus, other objects could not instantiate a `Cab` object.

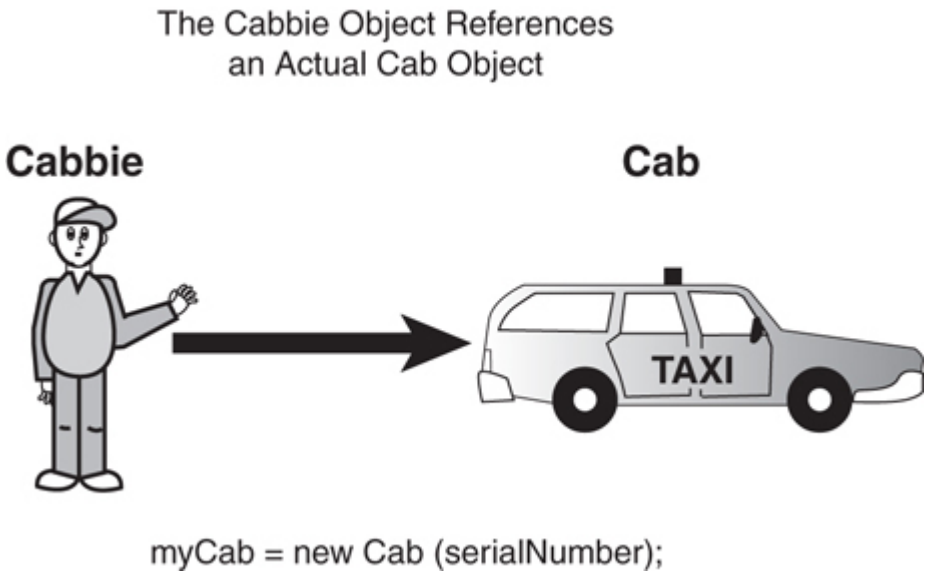

Figure 4.3 The `Cabbie` object referencing a cab object.

> ### Multiple Constructors
>
> It's worth noting that it is sometimes considered a less than ideal practice to use more than one constructor nowadays. With the prevalence of Inversion of Control (IoC) containers and the like, it's frowned upon, and even unsupported, for a number of frameworks without special configuration.

Accessors

In most, if not all, examples in this book, the attributes are defined as `private` so that any other objects cannot access the attributes directly. It would be ridiculous to create an object in isolation that does not interact with other objects—for we want to share appropriate information. That said, isn't it sometimes necessary to inspect and, perhaps, change another class's attribute? The answer is, of course, yes. There are many times when an object needs to access another object's attributes; however, it does not need to do it directly.

A class should be very protective of its attributes. For example, you do not want object `A` to have the capability to inspect or change the attributes of object `B` without object `B` having control. There are several reasons for this; the most important reasons boil down to data integrity and efficient debugging.

Assume that a bug exists in the `Cab` class. You have tracked the problem to the `Name` attribute. Somehow it is getting overwritten, and garbage is turning up in some name queries. If `Name` were `public` and any class could change it, you would have to go searching through all the possible code, trying to find places that reference and change `Name`. However, if you let only a `Cabbie` object change `Name`, you'd have to look only in a method of the `Cabbie` class. This access is provided by a type of method called an *accessor*. Sometimes accessors are referred to as getters and setters, and sometimes they're simply called `get()` and `set()`. By convention, in this book we name the methods with the `set` and `get` prefixes, as in the following:

```
// Set the Name of the Cabbie
public void setName(String iName) {
    name = iName;
}

// Get the Name of the Cabbie
public String getName() {
    return name;
}
```

In this code snippet, a `Supervisor` object must ask the `Cabbie` object to return its name (see Figure 4.4). The important point here is that the `Supervisor` object can't retrieve the information on its own; it must ask

the `Cabbie` object for the information. This concept is important at many levels. For example, you might have a `setAge()` method that checks to see whether the age entered was 0 or below. If the age is less than 0, the `setAge()` method can refuse to set this obviously incorrect value. In general, the setters are used to ensure a level of data integrity.

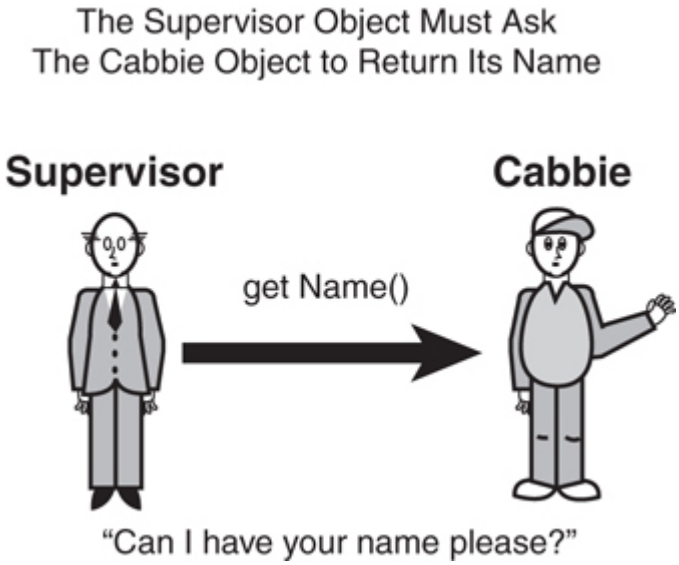

Figure 4.4 Asking for information.

This is also an issue of security. You may have sensitive data, such as passwords or payroll information, that you want to control access to. Thus, accessing data via getters and setters provides the capability to use mechanisms like password checks and other validation techniques. This greatly increases the integrity of the data.

> ### Objects
>
> Actually, there isn't a physical copy of each nonstatic method for each object. Each object would point to the same physical code. However, from a conceptual level, you can think of objects as being wholly independent and having their own attributes and methods.

The following code fragment illustrates how to define a static method, and Figure 4.5 shows how more than one object points to the same code.

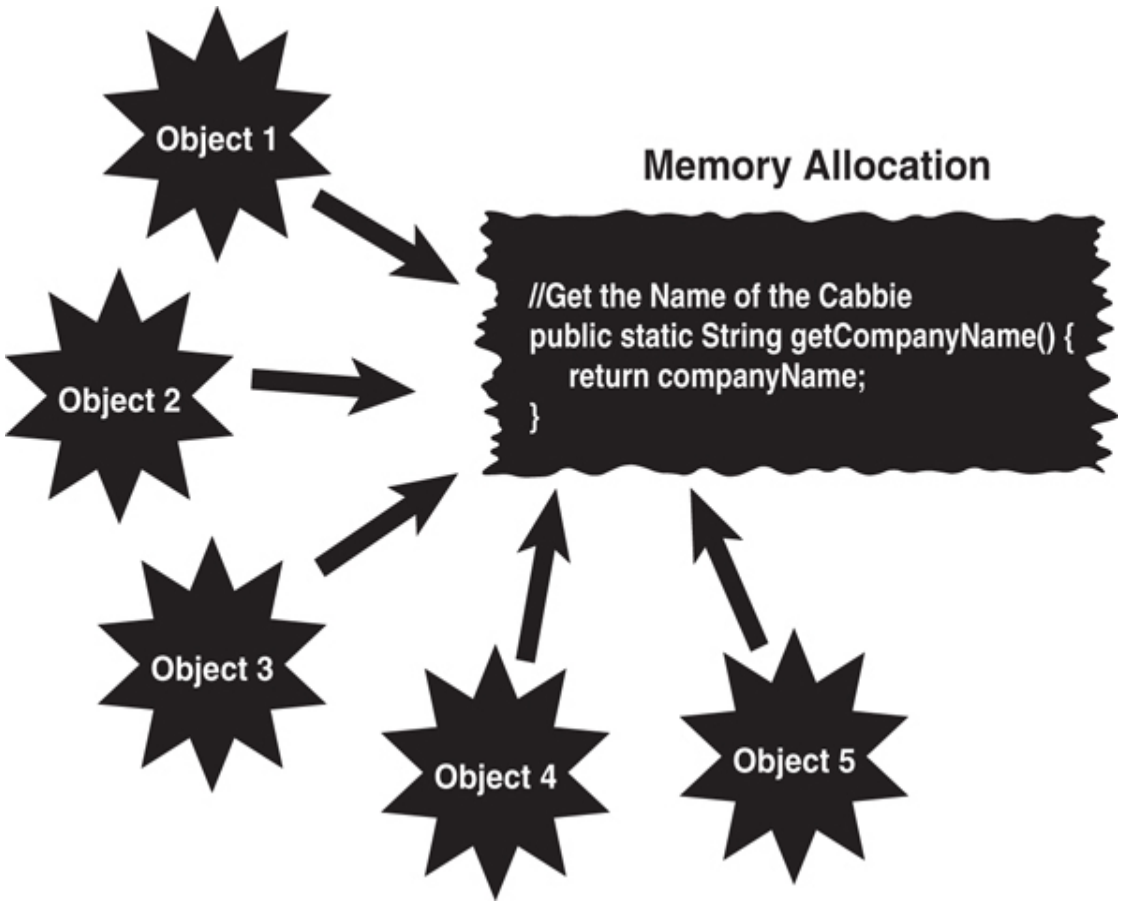

Figure 4.5 Method memory allocation.

> ### Static Attributes
>
> If an attribute is static, and the class provides a setter for that attribute, any object that invokes the setter will change the single copy. Thus, the value for the attribute will change for all objects.

Click here to view code image

```
// Get the Name of the Cabbie
public static String getCompanyName() {
    return companyName;
}
```

Notice that the `getCompanyName` method is declared as `static`, as a class method; class methods are described in more detail in Chapter 3. Remember that the attribute `companyName` is also declared as `static`. A

method, like an attribute, can be declared `static` to indicate that there is only one copy of the method for the entire class.

Public Interface Methods

Both the constructors and the accessor methods are declared as public and are part of the public interface. They are singled out because of their specific importance to the construction of the class. However, much of the *real* work is provided in other methods. As mentioned in Chapter 2, the public interface methods tend to be very abstract, and the implementation tends to be more concrete. For this class, we provide a method called `giveDestination` that is the public interface for the user to describe where she wants to go:

```
public void giveDestination(){

}
```

What is inside this method is not important at this time. The main point is that this is a public method, and it is part of the public interface to the class.

Private Implementation Methods

Although all the methods discussed so far in this chapter are defined as `public`, not all the methods in a class are part of the public interface. It is common for methods in a class to be hidden from other classes. These methods are declared as `private`:

```
private void turnRight(){
}

private void turnLeft() {
}
```

These private methods are meant to be part of the implementation and not the public interface. You might ask who invokes these methods, if no other class can. The answer is simple: You might have already surmised that these methods are called internally from the class itself. For example, these methods could be called from within the method `giveDestination`:

```
public void giveDestination(){

.. some code

    turnRight();
    turnLeft();

.. some more code

}
```

As another example, you may have an internal method that provides encryption that you will use only from within the class itself. In short, this encryption method can't be called from outside the instantiated object itself.

The point here is that private methods are strictly part of the implementation and are not accessible by other classes.

Conclusion

In this chapter we have gotten inside a class and described the fundamental concepts necessary for understanding how a class is built. Although this chapter takes a practical approach to discussing classes, Chapter 5, "Class Design Guidelines," covers the class from a general design perspective.

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.

Tyma, Paul, Gabriel Torok, and Troy Downing. 1996. *Java Primer Plus*. Berkeley, CA: The Waite Group.

5. Class Design Guidelines

As we have already discussed, OO programming supports the idea of creating classes that are complete packages, encapsulating the data and behavior of a single entity. So, a class should represent a logical component, such as a taxicab.

This chapter presents several suggestions for designing classes. Obviously, no list such as this can be considered complete. You will undoubtedly add many guidelines to your personal list and incorporate useful guidelines from other developers.

Modeling Real-World Systems

One of the primary goals of object-oriented (OO) programming is to model real-world systems in ways similar to the ways in which people actually think. Designing classes is the object-oriented way to create these models. Rather than using a structured, or *top-down*, approach, where data and behavior are logically separate entities, the OO approach encapsulates the data and behavior into objects that interact with each other. We no longer think of a problem as a sequence of events or routines operating on separate data files. The elegance of this mindset is that classes literally model real-world objects and how these objects interact with other real-world objects.

These interactions occur in a way similar to the interactions between real-world objects, such as people. Thus, when creating classes, you should design them in a way that represents the true behavior of the object. Let's use the cabbie example from previous chapters. The `Cab` class and the `Cabbie` class model a real-world entity. As illustrated in Figure 5.1, the `Cab` and the `Cabbie` objects encapsulate their data and behavior, and they interact through each other's public interfaces.

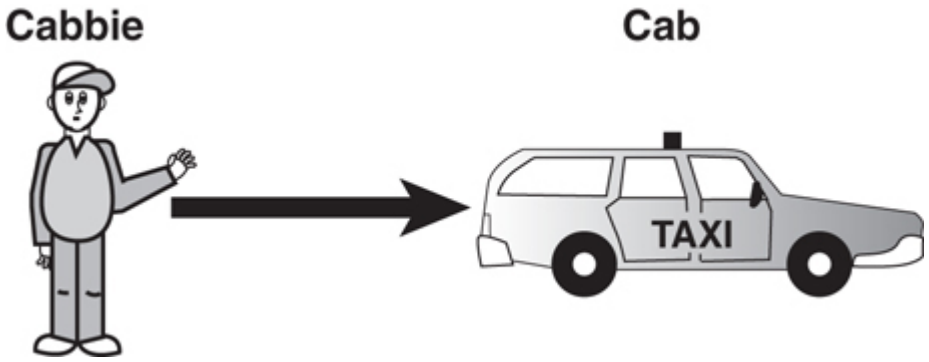

Figure 5.1 A cabbie and a cab are real-world objects.

When OO programming was first becoming popular, it was difficult for many structured programmers to make the transition. One primary mistake structured programmers made was to create a class that had behavior but no class data, in effect creating a set of functions or subroutines in the structured model. This was not desirable because it didn't take advantage of the power of encapsulation.

This is only partially true now. Currently, much development is done with anemic domain models, a.k.a. data transfer objects (DTOs) and view models that have just enough data to populate a view or just the right amount of data that is needed by a consumer. Much more focus has been placed on behaviors and operating on the data, and that is handled via interfaces. Encapsulating the behaviors into single-responsibility interfaces and coding to the interfaces keeps code flexible and modular and far easier to maintain.

> **Note**
>
> One of my favorite books pertaining to class design guidelines and suggestions remains *Effective C++: 50 Specific* Ways to Improve Your Programs and Designs, by Scott Meyers. It offers important information about program design in a very concise manner.

One of the reasons why *Effective C++* interests me so much is that, because C++ is backward compatible with C, you can write structured code in C++ without using OO design principles. As I mentioned earlier, during interviews, some people claim that they are OO programmers simply because they program in C++. This indicates a total misunderstanding of what OO design is all about. Thus, you may have to pay more attention to the

OO design issues in languages such as C++ as opposed to Java, Swift, or .NET.

Identifying the Public Interfaces

It should be clear by now that perhaps the most important issue when designing a class is to keep the public interface to a minimum. The entire purpose of building a class is to provide something useful and concise. In their book *Object-Oriented Design in Java*, Gilbert and McCarty state that "the interface of a well-designed object describes the services that the client wants accomplished." If a class does not provide a useful service to a user, it should not have been built in the first place.

The Minimum Public Interface

Providing the minimum public interface makes the class as concise as possible. The goal is to provide the user with the exact interface to do the job right. If the public interface is incomplete (that is, there is missing behavior), the user will not be able to do the complete job. If the public interface is not properly restricted (that is, the user has access to behavior that is unnecessary or even dangerous), problems can result in the need for debugging, and even trouble with system integrity and security can surface.

Creating a class is a business proposition, and as with all steps in the design process, it is very important that the users are involved with the design right from the start and throughout the testing phase. In this way, the utility of the class, as well as the proper interfaces, will be assured.

> ### Extending the Interface
>
> Even if the public interface of a class is insufficient for a certain application, object technology easily allows the capability to extend and adapt this interface. In short, if properly designed, a new class can utilize an existing class and create a new class with an extended interface.
>
> This is the point where, if you're adding behaviors, the developers should not be using inheritance,

To illustrate, consider the cabbie example again. If other objects in the system need to get the name of a cabbie, the `Cabbie` class must provide a public interface to return its name; this is the `getName()` method. Thus, if a `Supervisor` object needs a name from a `Cabbie` object, it must invoke the `getName()` method from the `Cabbie` object. In effect, the supervisor is asking the cabbie for its name (see Figure 5.2).

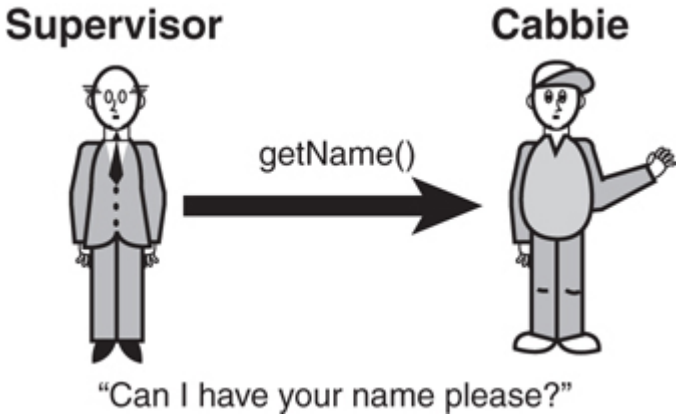

Figure 5.2 The public interface specifies how the objects interact.

Users of your code need to know nothing about its internal workings. All they need to know is how to instantiate and use the object. In short, provide users a way to get there but hide the details.

Hiding the Implementation

The need for hiding the implementation has already been covered in great detail. Whereas identifying the public interface is a design issue that revolves around the users of the class, the implementation should not involve the users at all. The implementation must provide the services that the user needs, but how these services are actually performed should not be made apparent to the user. A class is most useful if the implementation can change without affecting the users. Basically, a change to the implementation should not necessitate a change in the user's application code. Again, the best way to enable change of behaviors is via the use of interfaces and composition.

> ### Customer Versus User
>
> Sometimes I use the term *customer rather* than *user* when referring to the people who will actually be using the software. Users of the

> system may, in fact, be customers. In the same vein, users who are part of your organization can be called *internal customers*. This may seem like a trivial point, but I think it is important to think of all end users as actual customers—and you must satisfy their requirements.

In the cabbie example, the `Cabbie` class might contain behavior pertaining to how, or where, he eats lunch. However, the cabbie's supervisor does not need to know what the cabbie has for lunch. Thus, this behavior is part of the implementation of the `Cabbie` object and should not be available to other objects in this system (see Figure 5.3). Gilbert and McCarty state that the prime directive of encapsulation is that "all fields shall be private." In this way, none of the fields in a class are accessible from other objects.

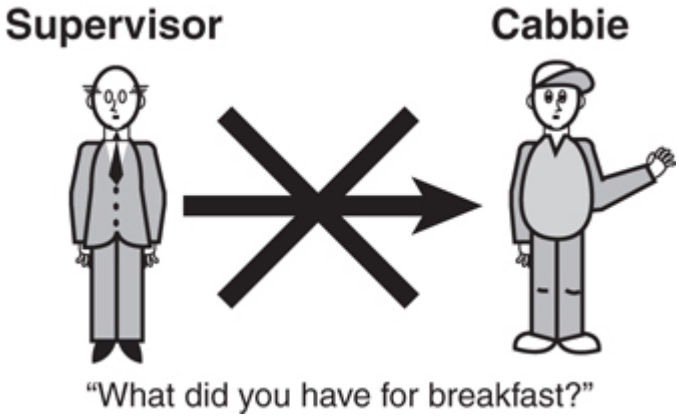

Figure 5.3 Objects don't need to know some implementation details.

Designing Robust Constructors (and Perhaps Destructors)

When designing a class, one of the most important design issues involves how the class will be constructed. Constructors are discussed in Chapter 3, "More Object-Oriented Concepts." Revisit this discussion if you need a refresher on guidelines for designing constructors.

First and foremost, a constructor should put an object into an initial, safe state. This includes issues such as attribute initialization and memory management. You also need to make sure the object is constructed properly in

the default condition. It is normally a good idea to provide a constructor to handle this default situation.

In languages that include destructors, it is of vital importance that the destructors include proper clean-up functions. In most cases, this clean-up pertains to releasing system memory that the object acquired at some point. Java and .NET reclaim memory automatically via a garbage collection mechanism. In languages such as C++, the developer must include code in the destructor to properly free up the memory that the object acquired during its existence. If this function is ignored, a memory leak will result.

> ### Constructor Injection
>
> This is a good point at which to introduce the concept of constructor injection, where service classes are injected on object creation (via a constructor) instead of within the class (using the new keyword). For example, the cabbie can get his license object, his radio information object (frequency, call sign, etc.), and the key that starts his cab passes into the object via a constructor.

> ### Memory Leaks
>
> When an object fails to properly release the memory that it acquired during an object's life cycle, the memory is lost to the entire operating system as long as the application that created the object is executing. For example, suppose multiple objects of the same class are created and then destroyed, perhaps in some sort of loop. If these objects fail to release their memory when they go out of scope, this memory leak slowly depletes the available pool of system memory. At some point, it is possible that enough memory will be consumed that the system will have no available memory left to allocate. This means that any application executing in the system would be unable to acquire any memory. This could put the application in an unsafe state and even lock up the system.

Designing Error Handling into a Class

As with the design of constructors, designing how a class handles errors is of vital importance. Error handling is discussed in detail in Chapter 3.

It is virtually certain that every system will encounter unforeseen problems. Thus, it is not a good idea to ignore potential errors. The developer of a good class (or any code, for that matter) anticipates potential errors and includes code to handle these conditions when they are encountered.

The general rule is that the application should never crash. When an error is encountered, the system should either fix itself and continue, or at minimum, exit gracefully without losing any data that's important to the user.

Documenting a Class and Using Comments

The topic of comments and documentation comes up in most programming books and articles, in every code review, and in every discussion you have about good design. Unfortunately, comments and good documentation are often not taken seriously, or even worse, they are ignored.

Most developers know that they should thoroughly document their code, but they don't usually want to take the time to do it. However, a good design is practically impossible without good documentation practices. At the class level, the scope might be small enough that a developer can get away with shoddy documentation. However, when the class gets passed to someone else to extend and/or maintain, or it becomes part of a larger system (which is what should happen), a lack of proper documentation and comments can undermine the entire system.

Many people have said all this before. One of the most crucial aspects of a good design, whether it's a design for a class or something else, is to carefully document the process. Implementations such as Java and .NET provide special comment syntax to facilitate the documentation process. Check out Chapter 4, "The Anatomy of a Class," for the appropriate syntax.

> ### Too Much Documentation
>
> Be aware that over-commenting can be a problem as well. Too much documentation and/or commenting can become background noise and may actually defeat the purpose of the documentation. Just like

> in good class design, make the documentation and comments straightforward and to the point. Well-written code is, in itself, the best documentation.

Building Objects with the Intent to Cooperate

We can safely say that almost no class lives in isolation. In most cases, there is little reason to build a class if it is not going to interact with other classes, unless the class will be used only once. This is a fact in the life of a class. A class will service other classes; it will request the services of other classes, or both. In later chapters we discuss various ways that classes interact with each other.

In the cabbie example, the cabbie and the supervisor are not standalone entities; they interact with each other at various levels (see Figure 5.4).

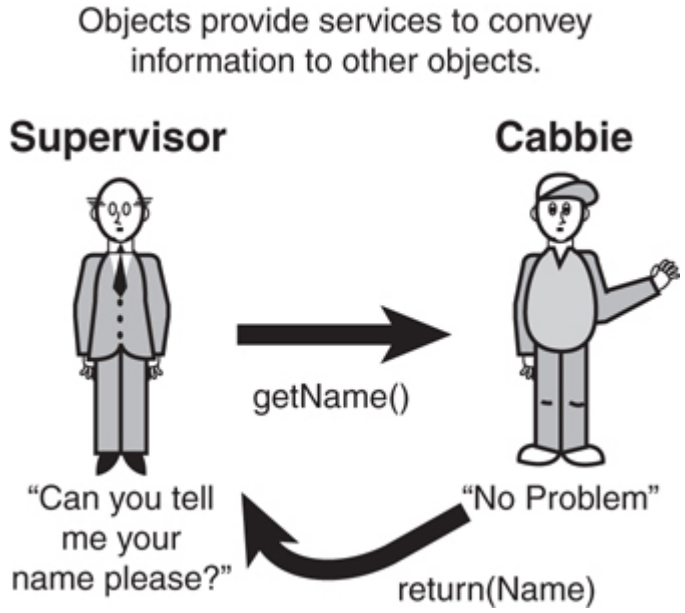

Figure 5.4 Objects should request information.

When designing a class, make sure you are aware of how other objects will interact with it.

Designing with Reuse in Mind

Objects can be reused in different systems, and code should be written with reuse in mind. For example, when a `Cabbie` class is developed and tested,

it can be used anywhere you need a cabbie. To make a class usable in various systems, the class must be designed with reuse in mind. This is where much of the thought is required in the design process. Attempting to predict all the possible scenarios in which a `Cabbie` object must operate is not a trivial task—in fact, it is virtually impossible.

Designing with Extensibility in Mind

Adding new features to a class might be as simple as extending an existing class, adding a few new methods, and modifying the behavior of others. It is not necessary to rewrite everything. This is where inheritance comes into play. If you have just written a `Person` class, you must consider the fact that you might later want to write an `Employee` class or a `Customer` class. Thus, having `Employee` inherit from `Person` might be the best strategy; in this case, the `Person` class is said to be *extensible*. You do not want to design `Person` so that it contains behavior that prevents it from being extended by classes such as `Employee` or `Customer` (assuming that in your design you really intend for other classes to extend `Person`). For example, you would not want to code functionality into an `Employee` class that is specific to supervisory functions. If you did, and then a class that does not require supervisory functionality inherited from `Employee`, you would have a problem.

This point touches on the abstraction guideline discussed earlier. `Person` should contain only the data and behaviors that are specific to a person. Other classes can then subclass it and inherit appropriate data and behaviors.

As we will cover in the SOLID discussion in Chapter 11, "Avoiding Dependencies and Highly Coupled Classes," and Chapter 12, "The SOLID Principles of Object-Oriented Design," classes should be open for extension but closed for modification. By using interfaces first and coding to them, you can use all sorts of patterns like Decorator to extend things without touching the code that's been tested and deployed live, for example.

> ### What Attributes and Methods Can Be Static?
>
> Static methods promote strong coupling to classes. You cannot abstract a static method. You cannot mock a static method or static

> class. You cannot provide a static interface. The only time it is reasonable to use static classes (within application development—framework development is a bit different) is if you're working with some sort of helper class or extension method that does not produce side effects. For example, a static class to add numbers is fine. A static class that interacts with a database or a web service is not.

Making Names Descriptive

Earlier we discussed the use of proper documentation and comments. Following a naming convention for your classes, attributes, and methods is a similar subject. There are many naming conventions, and the convention you choose is not as important as choosing one and sticking to it. However, when you choose a convention, make sure that when you create classes, attributes, and method names, you not only follow the convention but also make the names descriptive. When someone reads the name, he should be able to tell from the name what the object represents. These naming conventions are often dictated by the coding standards at various organizations.

> ### Good Naming
>
> Make sure that a naming convention makes sense. Often, people go overboard and create con-ventions that might make sense to them but are totally incomprehensible to others. Take care when forcing others to conform to a convention. Make sure that the conventions are sensible and that everyone involved understands the intent behind them. Make variables descriptive of their use, not encoded based on their type.

Making names descriptive is a good development practice that transcends the various development paradigms.

Abstracting Out Nonportable Code

If you are designing a system that must use nonportable (native) code (that is, the code will run only on a specific hardware platform), you should abstract this code out of the class. By abstracting out, we mean isolating the nonportable code in its own class or at least its own method (a method that

can be overridden). For example, if you are writing code to access a serial port of particular hardware, you should create a wrapper class to deal with it. Your class should then send a message to the wrapper class to get the information or services it needs. Do not put the system-dependent code into your primary class (see Figure 5.5).

Figure 5.5 A serial port wrapper.

For example, consider the situation when a programmer is interfacing directly with hardware. In these cases, the object code of the various platforms will most likely be quite different, and thus code must be written for each platform. However, if the functionality is placed in a wrapper class, then a user of the class can interface directly with the wrapper and not have to worry about the various low-level code. The wrapper class will deal with the differences in these platforms and decide which code to invoke.

Providing a Way to Copy and Compare Objects

Chapter 3 discussed the issue of copying and comparing objects. It is important to understand how objects are copied and compared. You might not want, or expect, a simple bitwise copy or compare operation. You must make sure that your class behaves as expected, and this means you have to spend some time designing how objects are copied and compared.

Keeping the Scope as Small as Possible

Keeping the scope as small as possible goes hand-in-hand with abstraction and hiding the implementation. The idea is to localize attributes and behaviors as much as possible. In this way, maintaining, testing, and extending a class are much easier. Using interfaces is a great way to enforce this.

> ### Scope and Global Data
>
> Minimizing the scope of global variables is a good programming style and is not specific to OO programming. Global variables are allowed in structured development, yet they can get dicey. In fact, there is no global data in OO development. Static attributes and methods are shared among objects of the same class; however, they are not available to objects not of the class. You could also share data via a file or database.

For example, if you have a method that requires a temporary attribute, keep it local. Consider the following code:

Click here to view code image

```
public class Math {

    int temp=0;

    public int swap (int a, int b) {

        temp = a;
        a=b;
        b=temp;

        return temp;

    }

}
```

What is wrong with this class? The problem is that the attribute `temp` is needed only within the scope of the `swap()` method. There is no reason for it to be at the class level. Thus, you should move `temp` within the scope of the `swap()` method:

Click here to view code image

```
public class Math {

    public int swap (int a, int b) {

        int temp=0;

        temp = a;
        a=b;
        b=temp;

        return temp;

    }

}
```

This is what is meant by keeping the scope as small as possible].

Designing with Maintainability in Mind

Designing useful and concise classes promotes a high level of maintainability. Just as you design a class with extensibility in mind, you should also design with future maintenance in mind.

The process of designing classes forces you to organize your code into many (ideally) manageable pieces. Separate pieces of code tend to be more maintainable than larger pieces of code (at least that's the idea). One of the best ways to promote maintainability is to reduce interdependent code—that is, changes in one class have no impact or minimal impact on other classes.

> ### Highly Coupled Classes
>
> Classes that are highly dependent on one another are considered highly coupled. Thus, if a change made to one class forces a change to another class, these two classes are considered highly coupled. Classes that have no such dependencies have a very low degree of coupling. For more information on this topic, refer to *The Object Primer*, by Scott Ambler.

If the classes are designed properly in the first place, any changes to the system should be made only to the implementation of an object. Changes to the public interface should be avoided at all costs. Any changes to the public interface will cause ripple effects throughout all the systems that use the interface.

For example, if a change were made to the `getName()` method of the `Cabbie` class, every single place in all systems that use this interface must be changed and recompiled. Finding all these method calls is a daunting task, and the likelihood of missing one is pretty high.

To promote a high level of maintainability, keep the coupling level of your classes as low as possible.

Using Iteration in the Development Process

As in most design and programming functions, using an iterative process is recommended. This dovetails well with the concept of providing minimal interfaces. Basically, this means *don't write all the code at once*! Create the code in small increments and then build and test it at each step. A good testing plan quickly uncovers any areas where insufficient interfaces are provided. In this way, the process can iterate until the class has the appropriate interfaces. This testing process is not simply confined to coding. Testing the design with walkthroughs and other design review techniques is very helpful. Testers' lives are more pleasant when iterative processes are used, because they are involved in the process early and are not simply handed a system that is thrown over the wall at the end of the development process.

Testing the Interface

The minimal implementations of the interface are often called *stubs*. (Gilbert and McCarty have a good discussion on stubs in *Object-Oriented Design in Java*.) By using stubs, you can test the interfaces without writing any *real* code. In the following example, rather than connecting to an actual database, stubs are used to verify that the interfaces are working properly (from the user's perspective—remember that interfaces are meant for the user). Thus, the implementation is not necessary at this point. In fact, it might cost

valuable time and energy to complete the implementation yet because the design of the interface will affect the implementation, and the interface is not yet complete.

In Figure 5.6, note that when a user class sends a message to the `DataBaseReader` class, the information returned to the user class is provided by code stubs and not by the actual database. (In fact, the database most likely does not exist yet.) When the interface is complete and the implementation is under development, the database can then be connected and the stubs disconnected.

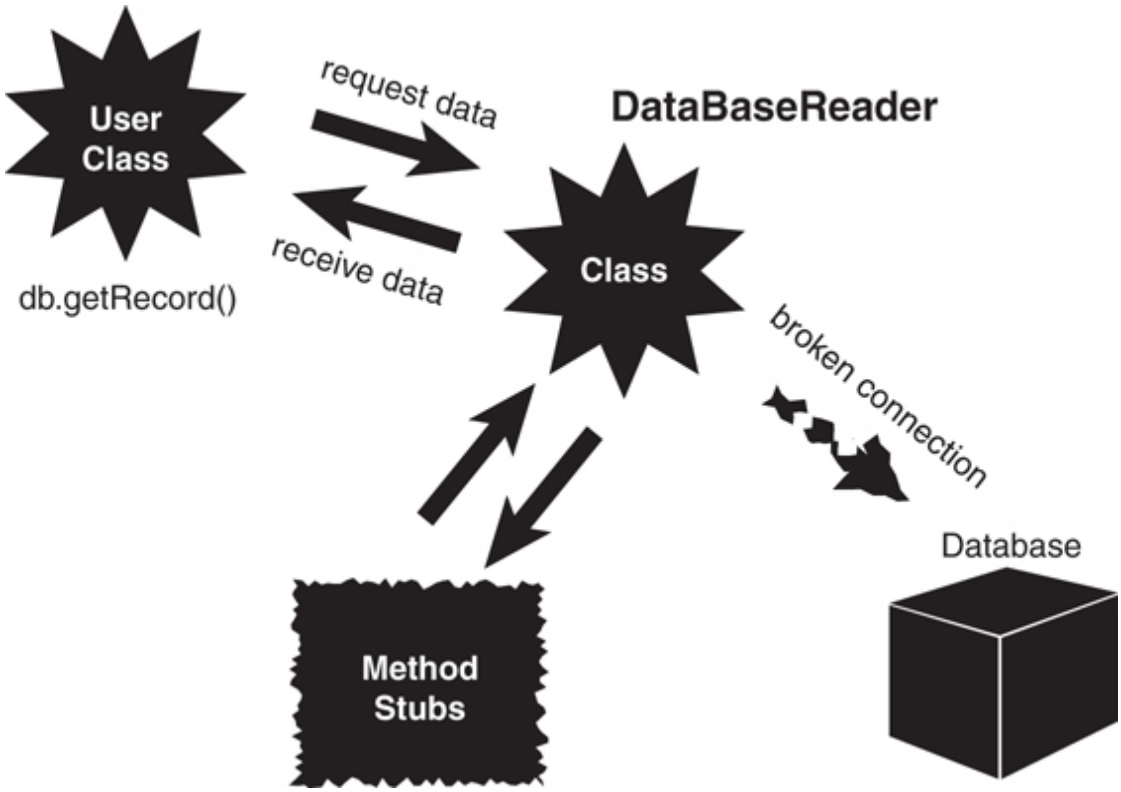

Figure 5.6 Using stubs.

Here is a code example that uses an internal array to simulate a working database (albeit a simple one):

Click here to view code image

```
public class DataBaseReader {

    private String db[] = {
"Record1","Record2","Record3","Record4","Record5"};
```

```
    private booleanDBOpen = false;
    private int pos;

    public void open(String Name){
        DBOpen = true;
    }

    public void close(){
        DBOpen = false;
    }

    public void goToFirst(){
        pos = 0;
    }

    public void goToLast(){
        pos = 4;
    }

    public int howManyRecords(){
        int numOfRecords = 5;
        return numOfRecords;
    }

    public String getRecord(int key){
        /* DB Specific Implementation */
        return db[key];
    }

    public String getNextRecord(){
        /* DB Specific Implementation */
        return db[pos++];
    }

}
```

Notice how the methods simulate the database calls. The strings within the array represent the records that will be written to the database. When the database is successfully integrated into the system, it will be substituted for the array.

> ### Keeping the Stubs Around
>
> When you are done with the stubs, don't delete them. Keep them in the code for possible use later—just make sure the users can't see

> them and the other team members know that they are there. In fact, in a well-designed program, your test stubs should be integrated into the design and kept in the program for later use. In short, design the testing right into the class! Perhaps even better, create stubs with mock data and coded to interfaces, and then you can swap them out with the actual implementation when the time comes.

As you find problems with the interface design, make changes and repeat the process until you are satisfied with the result.

Using Object Persistence

Object persistence is another issue that must be addressed in many OO systems. *Persistence* is the concept of maintaining the state of an object. When you run a program, if you don't save the object in some manner, the object dies, never to be recovered. These transient objects might work in some applications, but in most business systems, the state of the object must be saved for later use.

> ### Object Persistence
>
> Although the topic of object persistence and the topics in the next section might not be consid-ered true design guidelines, I believe that they must be addressed when designing classes. I introduce them here to stress that they must be addressed early on when designing classes.

In its simplest form, an object can persist by being serialized and written to a flat file. The state-of-the-art technology is now XML-based. Although it is true that an object theoretically can persist in memory as long as it is not destroyed, we will concentrate on storing persistent objects on some sort of storage device. There are three primary storage devices to consider:

- **Flat file system**—You can store an object in a flat file by serializing the object. This is definitely outdated. More often than not, objects are serialized to XML and/or JSON and written to some sort of file system or data store or web endpoint. They can be put into a

database or written to disk, which is the most common practice nowadays.

- **Relational database**—Some sort of middleware is necessary to convert an object to a relational model.
- **NoSQL database**—This may be a more efficient way to make objects persistent, but most companies have all their data in legacy systems and at this point in time are unlikely to convert their relational databases to OO databases. This is the most common form of a flexible structure database. MongoDB or Cosmos DB are two of the bigger names in this space.

Serializing and Marshaling Objects

We have already discussed the problem of using objects in environments that were originally designed for structured programming. The middleware example, where we wrote objects to a relational database, is one good example. We also touched on the problem of writing an object to a flat file or sending it over a network.

To send an object over a wire (for example, to a file, over a network), the system must deconstruct the object (flatten it out), send it over the wire, and then reconstruct it on the other end of the wire. This process is called *serializing* an object. The act of sending the object across a wire is called *marshaling* an object. A serialized object, in theory, can be written to a flat file and retrieved later, in the same state in which it was written.

The major issue here is that the serialization and deserialization must use the same specifications. It is sort of like an encryption algorithm. If one object encrypts a string, the object that wants to decrypt it must use the same encryption algorithm. Java provides an interface called `Serializable` that provides this translation.

This is another reason why data is separated from behaviors nowadays. It's far simpler to create an interface for a data contract and push that out to a web service than it is to make sure people have the same code on both sides.

Conclusion

This chapter presents many guidelines that can help you in designing classes. This is by no means a complete list of guidelines. You will undoubtedly come across additional guidelines as you go about your travels in OO design.

This chapter deals with design issues as they pertain to individual classes. However, we have already seen that a class does not live in isolation. Classes must be designed to interact with other classes. A group of classes that interact with each other is part of a system. Ultimately, these systems provide value to end users. Chapter 6, "Designing with Objects," covers the topic of designing complete systems.

References

Ambler, Scott. 2004. *The Object Primer*, Third Edition. Cambridge, United Kingdom: Cambridge University Press.

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

Jaworski, Jamie. 1997. *Java 1.1 Developers Guide*. Indianapolis, IN: Sams Publishing.

Jaworski, Jamie. 1999. *Java 2 Platform Unleashed*. Indianapolis, IN: Sams Publishing.

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

Tyma, Paul, Gabriel Torok, and Troy Downing. 1996. *Java Primer Plus*. Berkeley, CA: The Waite Group.

6. Designing with Objects

When you use a software product, you expect it to behave as advertised. Unfortunately, not all products live up to expectations. The problem is that when many products are produced, the majority of time and effort go into the engineering phase and not into the design phase.

Object-oriented (OO) design has been touted as a robust and flexible software development approach. The truth is that you can create both good and bad OO designs just as easily as you can create both good and bad non-OO designs. Don't be lulled into a false sense of security just because you are using a state-of-the-art design methodology. You must pay attention to the overall design and invest the proper amount of time and effort to create the best possible product.

In Chapter 5, "Class Design Guidelines," we concentrated on designing good classes. This chapter focuses on designing good systems. A *system* can be defined as classes that interact with each other. Proper design practices have evolved throughout the history of software development, and there is no reason you should not take advantage of the blood, sweat, and tears of your software predecessors, whether they used OO technologies or not.

Taking advantage of previous efforts is not limited to design practices; you can even incorporate existing legacy code in your object-oriented designs. In many cases, you can take code, which may have been working well for years, and literally wrap it in your objects. The *wrapping* is discussed later in the chapter.

Design Guidelines

One fallacy is that there is one true (best) design methodology. This is certainly not the case. There is no right or wrong way to create a design. Many design methodologies are available today, and they all have their

proponents. However, the primary issue is not which design method to use, but whether to use a method at all. This can be expanded beyond design to encompass the entire software development process. Some organizations do not follow a standard software development process, or they have one and don't adhere to it. The most important factor in creating a good design is to find a process that you and your organization feel comfortable with, stick to it, and keep refining it. It makes no sense to implement a design process that no one will follow.

Most books that deal with object-oriented technologies offer very similar strategies for designing systems. In fact, except for some of the object-oriented specific issues involved, much of the strategy is applicable to non-OO systems as well.

Generally, a solid OO design process includes the following steps:

1. Doing the proper analysis
2. Developing a statement of work that describes the system
3. Gathering the requirements from this statement of work
4. Developing a prototype for the user interface
5. Identifying the classes
6. Determining the responsibilities of each class
7. Determining how the various classes interact with each other
8. Creating a high-level model that describes the system to be built

For object-oriented development, the high-level system model is of special interest. The system, or object model, is made up of class diagrams and class interactions. This model should represent the system faithfully and be easy to understand and modify. We also need a notation for the model. This is where the Unified Modeling Language (UML) comes in. As you know, UML is not a design process but a modeling tool. In this book, I only use class diagrams within UML. I like to utilize class diagrams as a visual tool to assist with the

design process as well as with documentation—even if I don't use the other available UML tools.

> ### The Ongoing Design Process
>
> Despite the best intentions and planning, in all but the most trivial cases, the design is an ongoing process. Even after a product is in testing, design changes will pop up. It is up to the project manager to draw the line that says when to stop changing a product and adding features. I like to call this Version 1.

It is important to understand that many design methodologies are available. One early methodology, called the waterfall model, advocates strict boundaries between the various phases. In this case, the design phase is completed before the implementation phase, which is completed before the testing phase, and so on. In practice, the waterfall model has been found to be unrealistic. Currently, other design models, such as rapid prototyping, Extreme Programming, Agile, Scrum, and so on, promote a true iterative process. In these models, some implementation is attempted prior to completing the design phase as a type of proof-of-concept. Despite the recent aversion to the waterfall model, the goal behind the model is understandable. Coming up with a complete and thorough design before starting to code is a sound practice. You do not want to be in the release phase of the product and then decide to iterate through the design phase again. Iterating across phase boundaries is unavoidable; however, you should keep these iterations to a minimum (see Figure 6.1).

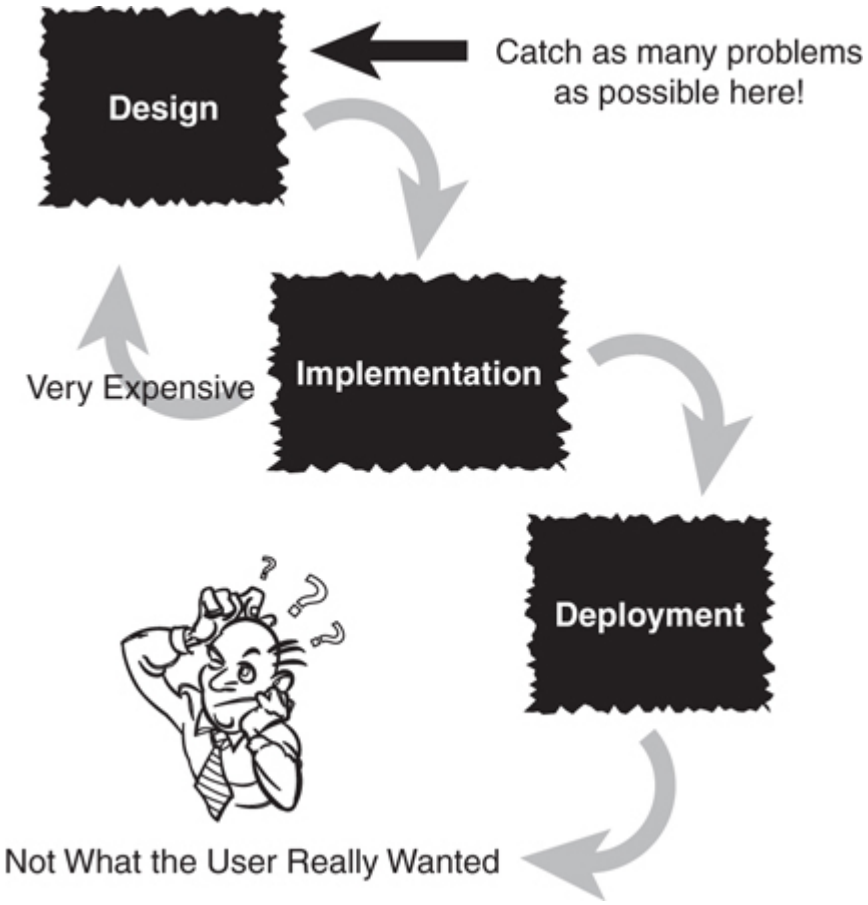

Figure 6.1 The waterfall method.

Simply put, the reasons to identify requirements early and keep design changes to a minimum are as follows:

- The cost of a requirement/design change in the design phase is relatively small.
- The cost of a design change in the implementation phase is significantly higher.
- The cost of a design change after the deployment phase is astronomical when compared to the first item.

Similarly, you would not want to start the construction of your dream house before the architectural design was complete. If I said that the Golden Gate Bridge or the Empire State Building was constructed without any consideration of design issues, you would consider the statement absolutely crazy. Yet, you would most likely not find it crazy if I told you that the

software you were using might contain some design flaws, and in fact, might not have been thoroughly tested.

In reality, it might be impossible to thoroughly test software, in the sense that absolutely *no* bugs exist. However, in theory, that is always the goal. We should always attempt to weed out as many bugs as possible. Bridges and software might not be directly comparable; however, software must strive for the same level of engineering excellence as the "harder" engineering disciplines such as bridge building. Poor-quality software can be lethal—it's not just wrong numbers on payroll checks. For example, inferior software in medical equipment can kill and maim people. Yet, you may be willing to live with having to reboot your computer every now and then. But the same cannot be said for a bridge failing.

> ### Safety Versus Economics
>
> Would you want to cross a bridge that has not been inspected and tested? Unfortunately, with many software packages, users are left with the responsibility of doing much of the testing. This is very costly for both the users and the software providers. Unfortunately, short-term economics often seem to be the primary factor in making project decisions.

Because customers seem to be willing to pay a limited price and put up with software of poor quality, some software providers find that it is cheaper in the long run to let the customers test the product rather than do it themselves. In the short term this might be true, but in the long run it costs far more than the software provider realizes. Ultimately, the software provider's reputation will be damaged.

Some computer software companies are willing to use the beta test phase to let the customers do testing—testing that should, theoretically, have been done before the beta version ever reached the customer. Many customers are willing to take the risk of using prerelease software because they are anxious to get the functionality the product promises. Conversely, some customers resist new releases like the plague. If it works, don't fix it. Upgrading can be a nightmare!

After the software is released, problems that have not been caught and fixed prior to release become much more expensive. To illustrate, consider the dilemma automobile companies face when they are confronted with a recall. If a defect in the automobile is identified and fixed before it is shipped (ideally before it is manufactured), it is much cheaper than if all delivered automobiles have to be recalled and fixed one at a time. Not only is this scenario very expensive, but it damages the reputation of the company. In an increasingly competitive market, high-quality software, support services, and reputation are *the* competitive advantage (see Figure 6.2).

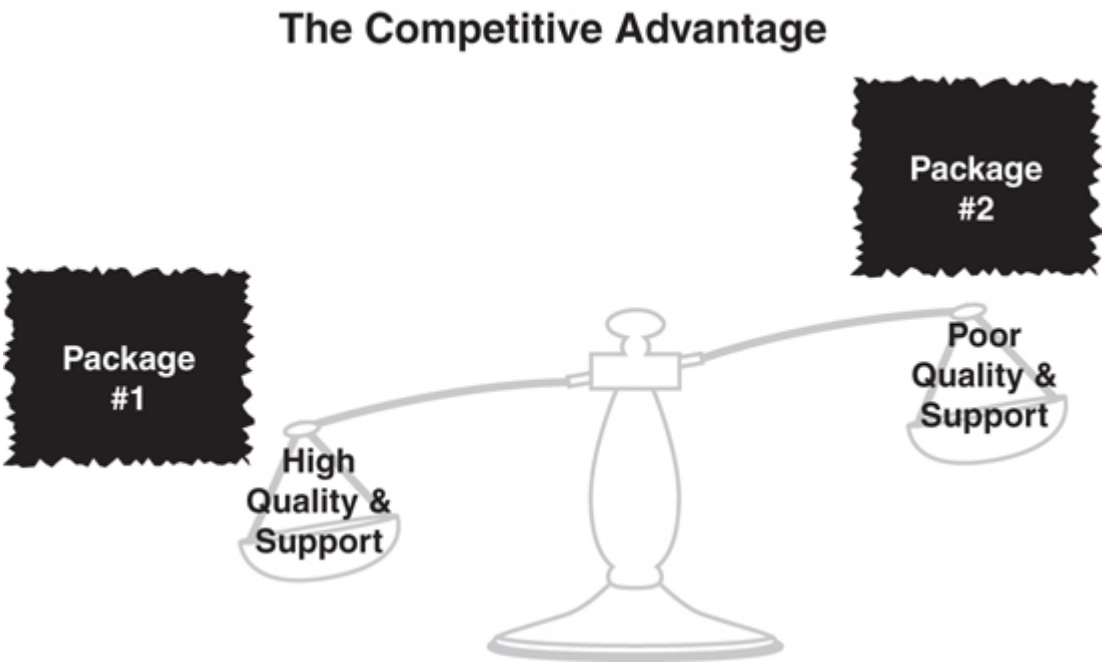

Figure 6.2 The competitive advantage.

The following sections provide brief summaries of the items listed previously as being part of the design process. Later in the chapter, we work through an example that explains in greater detail each of these items.

Performing the Proper Analysis

A lot of variables are involved in building a design and producing a software product. The users must work hand in hand with the developers at all stages. In the analysis phase, the users and the developers must do the proper research and analysis to determine the statement of work, the requirements of the project, and whether to actually do the project. The last point might seem a bit surprising, but it is important. During the analysis

phase, there must not be any hesitation to terminate the project if a valid reason exists to do so. Too many times, pet project status or some political inertia keeps a project going, regardless of the obvious warning signs that cry out for project cancellation. Assuming that the project is viable, the primary focus of the analysis phase is for everyone to learn the systems (both the old and the proposed new one) and determine the system requirements.

> ### Generic Software Principles
>
> Most of these practices are not specific to OO. They apply to software development in general.

Developing a Statement of Work

The *statement of work* (SOW) is a document that describes the system. Although determining the requirements is the ultimate goal of the analysis phase, at this point the requirements are not yet in a final format. The SOW should give anyone who reads it a complete, high level understanding of the system. Regardless of how it is written, the SOW must represent the complete system and be clear about how the system will look and feel.

The SOW contains everything that must be known about the system. Many customers create a *request for proposal* (RFP) for distribution, which is similar to the statement of work. A customer creates an RFP that completely describes the system the customer wants built and releases it to multiple vendors. The vendors then use this document, along with whatever analysis they need to do, to determine whether they should bid on the project, and if so, what price to charge.

Gathering the Requirements

The *requirements document* describes what the users want the system to do. Even though the level of detail of the requirements document does not need to be of a highly technical nature, the requirements must be specific enough to represent the true nature of the user's needs for the end product. The requirements document must be of sufficient detail for the user to make educated judgments about the completeness of the system. It must also be of

specific detail for a design group to use the document to proceed with the design phase.

Whereas the SOW is a document written in paragraph (even narrative) form, the requirements are usually represented as a summary statement or presented as bulleted items. Each individual bulleted item represents one specific requirement of the system. The requirements are distilled from the statement of work. This process is shown later in the chapter.

In many ways, these requirements are the most important part of the system. The SOW might contain irrelevant material; however, the requirements are the final representation of the system that must be implemented. All future documents in the software development process will be based on the requirements.

Developing a System Prototype

One of the best ways to make sure users and developers understand the system is to create a *prototype*. A prototype can be just about anything; however, most people consider the prototype to be a simulated user interface. By creating actual screens and screen flows, it is easier for people to get an idea of what they will be working with and what the system will feel like. In any event, a prototype will almost certainly not contain all the functionality of the final system.

Most prototypes are created with an integrated development environment (IDE). However, in some basic cases, drawing the screens on a whiteboard or even on paper might be all that is needed. Remember that you are not necessarily creating business logic (the logic/code behind the interface that actually does the work) when you build the prototype, although it is possible to do so. The look and feel of the user interface is a major concern at this point. Having a good prototype can help immensely when identifying classes.

Identifying the Classes

After the requirements are documented, the process of identifying classes can begin. From the requirements, one straightforward way of identifying classes is to highlight all the nouns. These tend to represent objects, such as people,

places, and things. Don't be too fussy about getting all the classes right the first time. You might end up eliminating classes, adding classes, and changing classes at various stages throughout the design. It is important to get something down first. Take advantage of the fact that the design is an iterative process. As in other forms of brainstorming, get something down initially, with the understanding that the final result might look nothing like the initial pass.

Determining the Responsibilities of Each Class

You need to determine the responsibilities of each class you have identified. This includes the data that the class must store and what operations the class must perform. For example, an `Employee` object would be responsible for calculating payroll and transferring the money to the appropriate account. It might also be responsible for storing the various payroll rates and the account numbers of various banks.

Determining How the Classes Collaborate with Each Other

Most classes do not exist in isolation. Although a class must fulfill certain responsibilities, many times it will have to interact with another class to get something it wants. This is where the messages between classes apply. One class can send a message to another class when it needs information from that class, or if it wants the other class to do something for it.

Creating a Class Model to Describe the System

When all the classes are determined and the class responsibilities and collaborations are listed, a class model that represents the complete system can be constructed. The class model shows how the various classes interact within the system.

In this book, we are using UML to model the system. Several tools on the market use UML and provide a good environment for creating and maintaining UML class models. As we develop the example in the next section, we will see how the class diagrams fit into the big picture and how

modeling large systems would be virtually impossible without some sort of good modeling notation and modeling tool.

Prototyping the User Interface in Code

During the design process, we must create a prototype of our user interface. This prototype will provide invaluable information to help navigate through the iterations of the design process. As Gilbert and McCarty in *Object-Oriented Design in Java* aptly point out, "to a system user, the user interface is the system." There are several ways to create a user interface prototype. You can sketch the user interface by drawing it on paper or a whiteboard. You also can use a special prototyping tool, or even a language environment like Visual Basic, which is often used for rapid prototyping. Or you can use the IDE from your favorite development tool to create the prototype. However, at this point they are basically facades; the business logic is not necessarily in place.

However you develop the user interface prototype, make sure that the users have the final say on the look and feel.

Object Wrappers

Several times in the previous chapters I have indicated that one of my primary goals in this book is to dispel the fallacy that object-oriented programming is a separate paradigm from structured programming, and is even at odds with it. In fact, as I have already mentioned, I am often asked the following question: "Are you an object-oriented programmer or a structured programmer?" The answer is always the same—I am both!

In my mind, there is no way to write a program without using structures. Thus, when you write a program that uses an object-oriented programming language and are using sound object-oriented design techniques, you are also using structured programming techniques. There is no way around this.

For example, when you create a new object that contains attributes and methods, those methods will include structured code. In fact, I might even say that these methods will contain *mostly* structured code. This approach fits in well with the container concept that we have encountered in earlier chapters.

In fact, when I get to the point where I am coding at the method level, my coding thought process hasn't changed much since when I was programming in structured languages, such as Cobol, C, and the like. This is not to say that it is exactly the same, because I obviously have had to adjust to some object-oriented constructs; however, the fundamental approach to coding at the method level is virtually the same as programming has always been.

Now I'll return to the question "Are you an object-oriented programmer or a structured programmer?" I often like to say that programming is programming. By this I contend that being a good programmer means understanding the basics of programming logic and having a passion for coding. Often you will see ads for a programmer with a specific skill set—let's say a specific language like Java.

Although I totally understand that an organization may well need an experienced Java programmer in a pinch, over the long run I would prefer to focus on hiring a programmer who has a wide range of programming experience and who can learn and adjust quickly when new technologies emerge. Some of my colleagues do not always agree with this; however, I believe that when hiring, I look more at what a potential employee can learn than what they already know. The passion part is critical because it ensures that an employee will always be exploring new technologies and development methodologies.

Structured Code

Although the basics of programming logic may be debated, as I have stressed, the fundamental object-oriented constructs are *encapsulation*, *inheritance*, *polymorphism*, and *composition*. In most textbooks that I have seen, the basic constructs of structured programming are *sequence*, *conditions*, and *iterations*.

The sequence part is a given, because it seems logical to start at the top and proceed in a logical manner to the bottom. For me, the meat of structured programming resides in the conditions and iterations, which I call if-statements and loops, respectively.

Take a look at the following Java code that starts at 0 and loops 10 times, printing out the value if it equals 5:

Click here to view code image

```
class MainApplication {

    public static void main(String args[]) {

        int x = 0;

        while (x <= 10) {

            if (x==5) System.out.println("x = " + x);
            x++;
        }
    }
}
```

Now while this code is written in an object-oriented language, the code that resides inside the main method is structured code. All three basics of structured programming are present: *sequence*, *conditions*, and *iterations*.

The sequence part is easy to identify because the first line executed is

```
int x = 0;
```

When that line completes, the next line is executed:

```
while (x <= 10) {
```

And so on. In short, this is tried and true top-down programming: start at the first line, execute it, and then go on to the next.

There is also a condition present in this code as part of the if-statement:

```
if (x==5)
```

Finally, there is a loop to complete the structured trio.

```
while (x <= 10) {
}
```

Actually, the while loop also contains a condition:

```
(x <= 10)
```

You can pretty much code anything with just these three constructs. In fact, the concept of the wrapper is basically the same for structured programming as it is for object-oriented programming. In structured design you wrap the code in functions (such as the main method in this example), and in object-oriented design you wrap the code in objects and methods.

Wrapping Structured Code

Although defining attributes is considered coding (for example, creating an integer), the behavior of an object resides in the methods. And these methods are where the bulk of the code logic is found.

Consider Figure 6.3. As you can see, an object contains methods, and these methods contain code, which can be anything from variable declarations to conditions to loops.

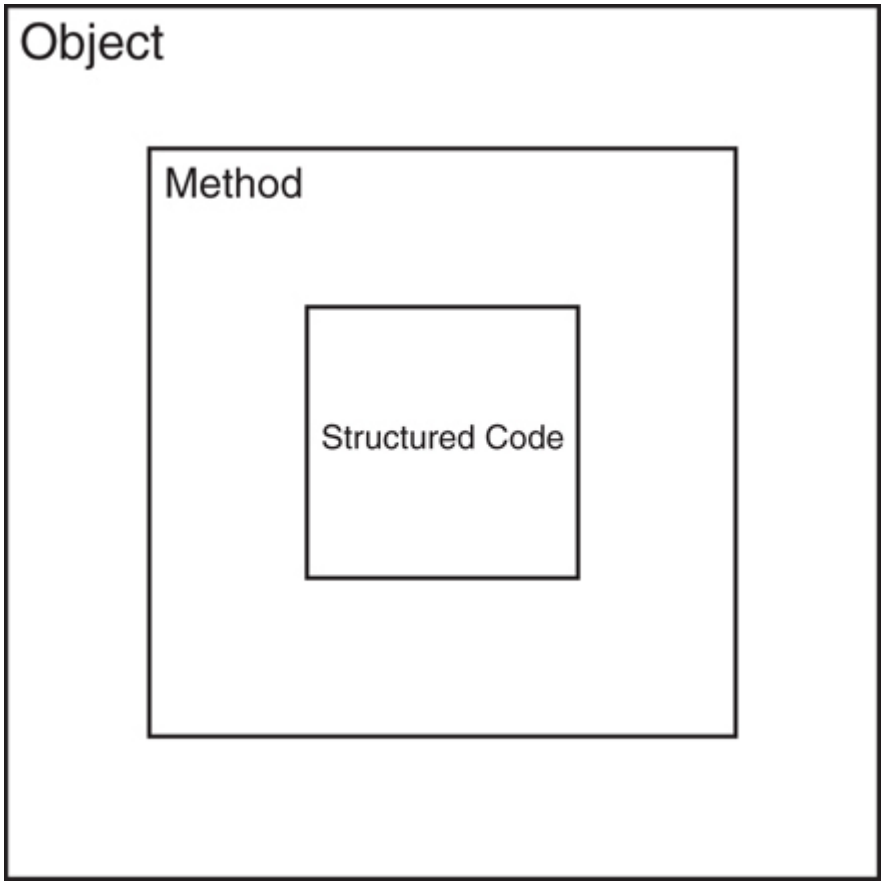

Figure 6.3 Wrapping structured code.

Let's consider a simple example in which we are wrapping the functionality for addition. Here we create a method named `add`, which accepts two integer parameters and returns their sum.

```
class SomeMath {

    public int add(int a, int b) {

        return a + b;

    }

}
```

As you can see, the structured code used to perform the addition (a + b) is wrapped inside the `add` method. Although this is a trivial example, that is all there is to wrapping structured code. Thus, when the user wants to use this method, all that is needed is the signature of the method as seen next:

Click here to view code image

```
public class TestMath {

    public static void main(String[] args) {

        int x = 0;

        SomeMath math = new SomeMath();
        x = math.add(1,2);
        System.out.println("x = " + x);
    }
}
```

Finally, we can add some more functionality that is a bit more interesting and complicated. Suppose that we wanted to include a method to calculate the Fibonacci value of a number. We can then `add` a method like this:

```
public static int fib(int n) {

    if (n < 2) {
        return n;
    } else {
        return fib(n-1)+fib(n-2);
    }
}
```

The whole point here is to show that we have an object-oriented method that contains (wraps) structured code, because the `fib` method contains conditions, recursion, and so on. And as mentioned in the introduction, it is possible to incorporate existing legacy code in wrappers as well.

Wrapping Nonportable Code

One other use of object wrappers is for the hiding of nonportable (or native) code. The concept is essentially the same; however, in this case the point is to take code that can be executed on only one platform (or a few platforms) and encapsulate it in a method providing a simple interface for the programmers using the code.

Consider the task of making the computer make a noise—in this case, a beep. On a Windows platform we can execute a beep with the following code:

```
System.out.println("\007");
```

Rather than making the programmer memorize the code (or look it up), you can provide a class called `Sound` that contains a method called `beep` as shown next:

```
class Sound {

    public void beep() {
        System.out.println("\007");
    }
}
```

Now, rather than having to know the code for making the sound, the programmer can use the class and call the `beep` method:

Click here to view code image

```
public class TestBeep {

    public static void main(String[] args) {
        Sound mySound = new Sound();
        mySound.beep();
    }
}
```

Not only is this simpler for the programmer to use, but you can extend the functionality of the class to include other sounds. Perhaps more importantly, when the code is used on a non-Windows platform, the interface for the user remains the same. In short, the team that builds the code for the `Sound` class will have to deal with the change in platform. For the programmers who utilize the class in their applications, the change will be seamless because they will still call the `beep` method.

Wrapping Existing Classes

Although the need to wrap legacy structured code, or even nonportable code, into a new (object-oriented) class may seem reasonable, the need to wrap existing classes might not seem so obvious. However, there are also many reasons to create wrappers for existing (object-oriented) classes.

Software developers often utilize code written by someone else. Perhaps the code was purchased from a vendor or even written internally within the same organization. In many of these cases, the code cannot be changed. Perhaps the individual who wrote the code is no longer with the organization, or the vendor cannot perform maintenance updates, and so on. This is where the true power of wrappers emerges.

The idea is to take an existing class and alter its implementation or interface by wrapping it inside a new class—just like we did for the structured code and nonportable code. The difference in this case is that, rather than putting an object-oriented face to the code, we are altering its implementation or interface.

Why would we want to do this? Well, the answer lies with both the implementation and the interface.

Consider the database example that we used in Chapter 2, "How to Think in Terms of Objects." Our goal was to provide the same interface for the developers regardless of which database they were using. In fact, if we need to support another database, our goal would remain the same—to make the transition to the new database transparent to the user (see Figure 2.3 as shown in Chapter 2).

Also, remember our earlier discussion about creating middleware to provide an interface between objects and relational databases. As developers, we want to use objects. Thus, we want functionality that will allow us to persist objects to a database. What we don't want to have to do is write SQL code for every single object transaction performed to a relational database. This is where we can consider middleware to be a wrapper, and many object-relational mapping products are available.

Conceptually, for me, the ultimate example of the interface and implementation paradigm is the discussion that we had regarding the power plant example in Chapter 2 and shown in Figure 2.1. In this case we can swap out (wrap) both: We can alter the interface by changing the outlet, and we can alter the implementation by changing the power generation facility.

The use of wrappers in software development is fairly extensive, not only from a developer's perspective but also from a vendor's. Wrappers are an important tool when developing software systems.

In this chapter, we have focused on various design considerations, including writing new code as well as utilizing previously written code, whether in house or from vendors. In some cases, wrappers are even design paradigms unto themselves. Design patterns, for example, utilize wrappers in various cases. As we will see later, the Decorator pattern focuses on wrapping the implementation, whereas the Adaptor pattern focuses on altering the interface. The discussion of design patterns is explored in more detail in Chapter 10, "Design Patterns."

Conclusion

This chapter covers the design process for complete systems. It is important to note that object-oriented and structured code are not mutually exclusive. In fact, you can't create objects without using structured code. Thus, while building object-oriented systems, you are also using structured techniques in the design.

Object wrappers are used to encapsulate many types of functionality, which can range from traditional structured (legacy) and object-oriented (classes) code to nonportable (native) code. The primary purpose of object wrappers

is to provide consistent interfaces for the programmers who are using the code.

In the next several chapters, we explore in more detail the relationships between classes. Chapter 7, "Mastering Inheritance and Composition," covers the concepts of inheritance and composition and how they relate to each other.

References

Ambler, Scott. 2004. *The Object Primer*, Third Edition. Cambridge, United Kingdom: Cambridge University Press.

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

Jaworski, Jamie. 1999. *Java 2 Platform Unleashed*. Indianapolis, IN: Sams Publishing.

Jaworski, Jamie. 1997. *Java 1.1 Developers Guide*. Indianapolis, IN: Sams Publishing.

McConnell, Steve. 2004. *Code Complete: A Practical Handbook of Software Construction*, Second Edition. Redmond, WA: Microsoft Press.

Weisfeld, Matt, and John Ciccozzi. September, 1999. "Software by Committee," *Project Management Journal* v5, number 1: 30–36.

Wirfs-Brock, R., B. Wilkerson, and L. Weiner. 1990. *Designing Object-Oriented Software*. Upper Saddle River, NJ: Prentice-Hall.

7. Mastering Inheritance and Composition

Inheritance and composition play major roles in the design of object-oriented (OO) systems. In fact, many of the most difficult and interesting design decisions come down to deciding between inheritance and composition.

These decisions have become much more interesting over the years as object-oriented design practices have evolved. Perhaps one of the most interesting debates revolves around inheritance. Although inheritance is one of the fundamental constructs of object-oriented development (a language must support inheritance to be considered object-oriented), some developers are even turning away from inheritance by implementing designs solely with composition.

It is common to use interface inheritance rather than direct inheritance for behaviors (implementing versus inheriting). Inheritance tends to be used often for data/models whereas implementation tends to be used for behaviors.

Regardless, both inheritance and composition are mechanisms for reuse. *Inheritance*, as its name implies, involves inheriting attributes and behaviors from other classes, where there is a true parent/child relationship. The child (or subclass) inherits directly from the parent (or superclass).

Composition, also as its name implies, involves building objects by using other objects. In this chapter we explore the obvious and subtle differences between inheritance and composition. Primarily, we will consider the appropriate times to use one or the other.

Reusing Objects

Perhaps the primary reason why inheritance and composition exist is object reuse. In short, you can build classes (which ultimately become objects) by utilizing other classes via inheritance and composition, which in effect are the only ways to reuse previously built classes.

Inheritance represents the is-a relationship that was introduced in Chapter 1, "Introduction to Object-Oriented Concepts." For example, a dog *is a* mammal.

Composition involves using other classes to build more complex classes—a sort of assembly. No parent/child relationship exists in this case. Basically, complex objects are composed of other objects. Composition represents a has-a relationship. For example, a car *has an* engine. Both the engine and the car are separate, potentially standalone objects. However, the car is a complex object that contains (has an) engine object. In fact, a child object might itself be composed of other objects; for example, the engine might include cylinders. In this case an engine *has a* cylinder, actually several.

When OO technologies first entered the mainstream, inheritance was often the first example used in how to design an OO system. That you could design a class once and then inherit functionality from it was considered one of the foremost advantages to using OO technologies. Reuse was the name of the game, and inheritance was the ultimate expression of reuse.

However, over time the luster of inheritance has dulled a bit. In fact, in some discussions, the use of inheritance itself is questioned. In their book *Java Design*, Peter Coad and Mark Mayfield have a complete chapter titled "Design with Composition Rather Than Inheritance." Many early object-based platforms did not even support true inheritance. As Visual Basic evolved into Visual Basic .NET, early object-based implementations did not include strict inheritance capabilities. Platforms such as the MS COM model were based on interface inheritance. Interface inheritance is covered in great detail in Chapter 8, "Frameworks and Reuse: Designing with Interfaces and Abstract Classes."

Today, the use of inheritance is still a major topic of debate. Abstract classes, which are a form of inheritance, are not directly supported in some languages, such as Objective-C and Swift. Interfaces are used even though they don't provide all the functionality that abstract classes do.

The good news is that the discussions about whether to use inheritance or composition are a natural progression toward some seasoned middle ground. As in all philosophical debates, there are passionate arguments on both sides. Fortunately, as is normally the case, these heated discussions have led to a more sensible understanding of how to utilize the technologies.

We will see later in this chapter why some people believe that inheritance should be avoided, and composition should be the design method of choice. The argument is fairly complex and subtle. In actuality, both inheritance and composition are valid class design techniques, and they each have a proper place in the OO developer's toolkit. And, at least, you need to understand both to make the proper design choice—not to mention maintenance of legacy code.

The fact that inheritance is often misused and overused is more a result of a lack of understanding of what inheritance is all about than a fundamental flaw in using inheritance as a design strategy.

The bottom line is that inheritance and composition are both important techniques in building OO systems. Designers and developers need to take the time to understand the strengths and weaknesses of both and to use each in the proper contexts.

Inheritance

Inheritance was defined in Chapter 1 as a system in which child classes inherit attributes and behaviors from a parent class. However, there is more to inheritance, and in this chapter we explore inheritance in greater detail.

Chapter 1 states that you can determine an inheritance relationship by following a simple rule: If you can say that Class B *is a* Class A, then this relationship is a good candidate for inheritance.

> ### Is-a
>
> One of the primary rules of OO design is that public inheritance is represented by an is-a relationship. In the case of interfaces you might add "behaves like" (implements). The data (attributes) that are

> inherited are the "is," the interfaces describing encapsulated behaviors are "acts like," and composition is "has a." The lines get pretty blurry, however.

Let's revisit the mammal example used in Chapter 1. Let's consider a `Dog` class. A dog has several behaviors that make it distinctly a dog, as opposed to a cat. For this example, let's specify two: A dog barks and a dog pants. So we can create a `Dog` class that has these two behaviors, along with two attributes (see Figure 7.1).

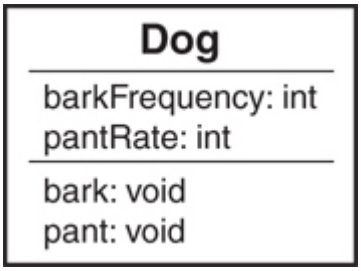

Figure 7.1 A class diagram for the `Dog` class.

Now let's say that you want to create a `GoldenRetriever` class. You could create a brand-new class that contains the same behaviors that the `Dog` class has. However, we could make the following, and quite reasonable, conclusion: A Golden Retriever is-a dog. Because of this relationship, we can inherit the attributes and behaviors from `Dog` and use it in our new `GoldenRetriever` class (see Figure 7.2).

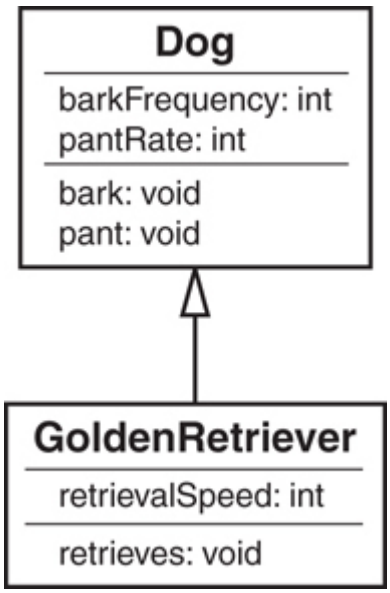

Figure 7.2 The `GoldenRetriever` class inherits from the `Dog` class.

The `GoldenRetriever` class now contains its own behaviors as well as all the more general behaviors of a dog. This provides us with some significant benefits. First, when we wrote the `GoldenRetriever` class, we did not have to reinvent part of the wheel by rewriting the `bark` and `pant` methods. Not only does this save some design and coding time, but it saves testing and maintenance time as well. The `bark` and `pant` methods are written only once and, assuming that they were properly tested when the `Dog` class was written, they do not need to be heavily tested again; but it does need to be retested because there are new interfaces, and so on.

Now let's take full advantage of our inheritance structure and create a second class under the `Dog` class: a class called `LhasaApso`. Whereas retrievers were bred for retrieving, Lhasa Apsos were bred for use as guard dogs. These dogs are not attack dogs; they have acute senses, and when they sense something unusual, they start barking. So we can create our `LhasaApso` class and inherit from the `Dog` class just as we did with the `GoldenRetriever` class (see Figure 7.3).

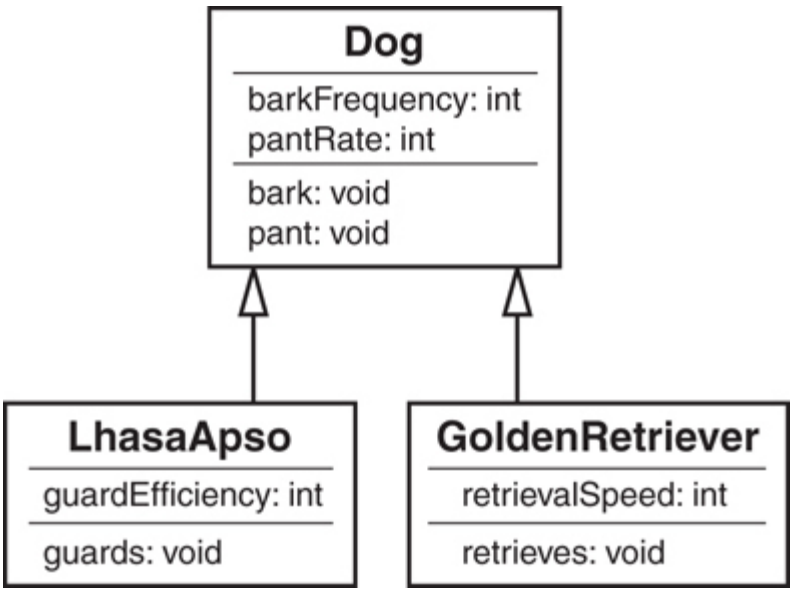

Figure 7.3 The `LhasaApso` class inherits from the `Dog` class.

Testing New Code

> In our example with the GoldenRetriever class, the bark and pant methods should be writ-ten, tested, and debugged when the Dog class is written. Theoretically, this code is now robust and ready to reuse in other situations. However, the fact that you do not need to rewrite the codedoes not mean it should not be tested. However unlikely, there might be some specific characteristic of a retriever that somehow breaks the code. The bottom line is that you should always test new code. Each new inheritance relationship creates a new context for using inher-ited methods. A complete testing strategy should take into account each of these contexts.

Another primary advantage of inheritance is that the code for `bark()` and `pant()` is in a single place. Let's say there is a need to change the code in the `bark()` method. When you change it in the `Dog` class, you do not need to change it in the `LhasaApso` class and the `GoldenRetriever` class.

Do you see a problem here? At this level the inheritance model appears to work very well. However, can you be certain that all dogs have the behavior contained in the `Dog` class?

In his book *Effective C++*, Scott Meyers gives a great example of a dilemma with design using inheritance. Consider a class for a bird. One of the most recognizable characteristics of a bird is, of course, that it can fly. So we create a class called `Bird` with a `fly` method. You should immediately understand the problem. What do we do with a penguin, or an ostrich? They are birds, yet they can't fly. You could override the behavior locally, but the method would still be called `fly`. And it would not make sense to have a method called `fly` for a bird that does not fly but only waddles, runs, or swims. This is an example of the Liskov Substitution Principle of SOLID, which we discuss in Chapter 12, "The SOLID Principles of Object-Oriented Design."

This leads to some potentially significant problems. For example, if a penguin has a `fly` method, the penguin might understandably decide to test it out. However, if the `fly` method was in fact overridden and the behavior to fly did not exist, the penguin would be in for a major surprise when the `fly` method is invoked after jumping over a cliff. Imagine the penguin's chagrin when the call to the `fly` method results in waddling instead of flight (or

even a no-op, which means no operation, where nothing happens at all). In this situation, waddling doesn't cut it. Just imagine if code such as this ever found its way into a spacecraft's guidance system.

In our dog example, we have designed the class so that all dogs have the ability to bark. However, some dogs do not bark. The Basenji breed is a barkless dog. Although these dogs do not bark, they do yodel. So should we reevaluate our design? What would this design look like? Figure 7.4 is an example that shows a more correct way to model the hierarchy of the `Dog` class.

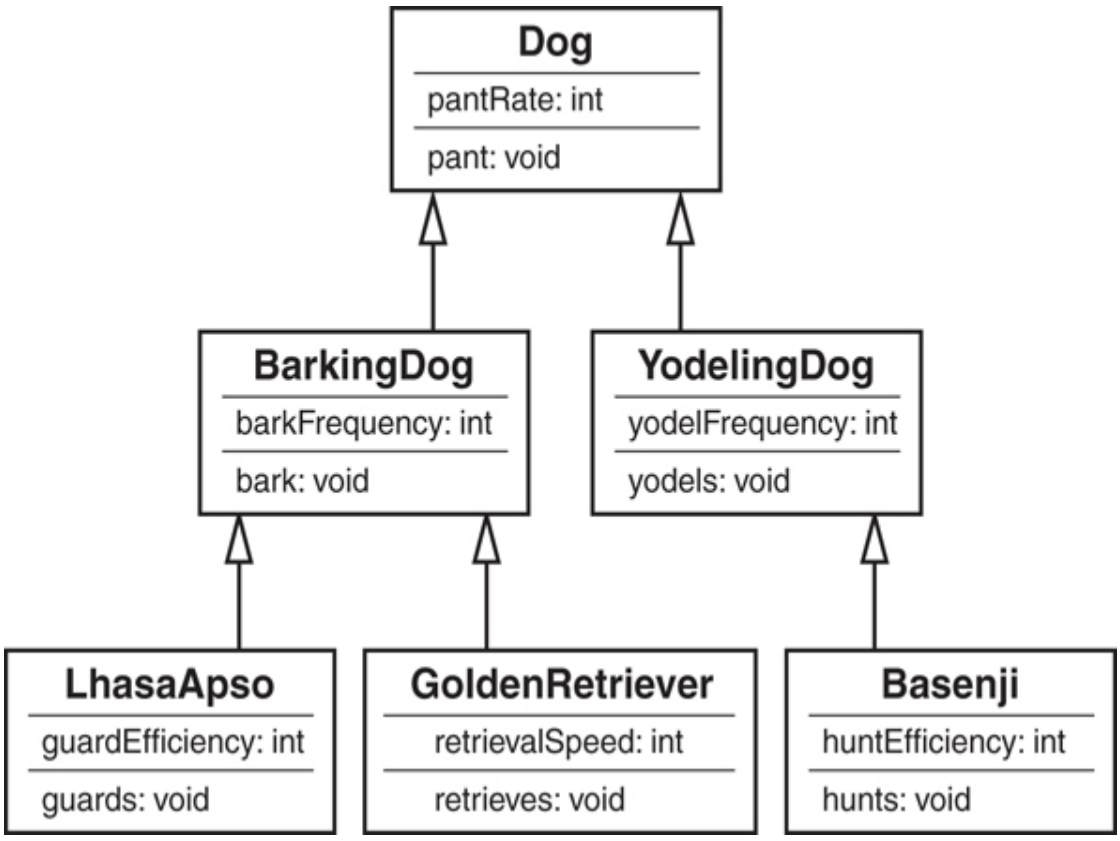

Figure 7.4 The `Dog` class hierarchy.

Generalization and Specialization

Consider the object model of the `Dog` class hierarchy. We started with a single class, called `Dog`, and we factored out some of the commonality between various breeds of dogs. This concept, sometimes called *generalization-specialization*, is yet another important consideration when using inheritance. The idea is that as you make your way down the

inheritance tree, things get more specific. The most general case is at the top of the tree. In our `Dog` inheritance tree, the class `Dog` is at the top and is the most general category. The various breeds—the `GoldenRetriever`, `LhasaApso`, and `Basenji` classes—are the most specific. The idea of inheritance is to go from the general to the specific by factoring out commonality.

In the `Dog` inheritance model, we started factoring out common behavior by understanding that although a retriever has some different behavior from that of a `LhasaApso`, the breeds do share some common behaviors—for example, they both pant and bark. Then we realized that all dogs do not bark—some yodel. Thus, we had to factor out the barking behavior into a separate `BarkingDog` class. The yodeling behavior went into a `YodelingDog` class. However, we realized that both barking dogs and barkless dogs still shared some common behavior—all dogs pant. Thus, we kept the `Dog` class and had the `BarkingDog` and the `YodelingDog` classes inherit from `Dog`. Now `Basenji` can inherit from `YodelingDog`, and `LhasaApso` and `GoldenRetriever` can inherit from `BarkingDog`.

We could have decided not to create two distinct classes for `BarkingDog` and `YodelingDog`. In this case we could implement all barking and yodeling as part of each individual breed's class—since each dog would sound differently. This is just one example of some of the design decisions that have to be made. Perhaps the best solution is to implement the barking and yodeling as interfaces, which we discuss in Chapter 8.

A design pattern, which is covered in Chapter 10, "Design Patterns," might be a good option in this case. A developer might not typically create these variations of `Dog`; they would either use a `Dog` (which implements `IDog`) or use a decorator to add behaviors to a `Dog` object.

Design Decisions

In theory, factoring out as much commonality as possible is great. However, as in all design issues, sometimes it really is too much of a good thing. Although factoring out as much commonality as possible might represent real life as closely as possible, it might not represent your model as closely as

possible. The more you factor out, the more complex your system gets. So you have a conundrum: Do you want to live with a more accurate model or a system with less complexity? You must make this choice based on your situation, for there are no hard guidelines to make the decision.

> ### What Computers Are Not Good At
>
> Obviously, a computer model can only approximate real-world situations. Computers are good at number crunching but are not as good at more abstract operations.

For example, breaking up the `Dog` class into `BarkingDog` and the `YodelingDog` models real life better than assuming that all dogs bark, but it does add a bit of complexity.

> ### Model Complexity
>
> At this level of our example, adding two more classes does not make things so complex that it makes the model untenable. However, in larger systems, when these kinds of decisions are made over and over, the complexity quickly adds up. In larger systems, keeping things as simple as possible is usually the best practice.

There will be instances in your design when the advantage of a more accurate model does not warrant the additional complexity. Let's assume that you are a dog breeder and that you contract out for a system that tracks all your dogs. The system model that includes barking dogs and yodeling dogs works fine. However, suppose that you do not breed any yodeling dogs—never have and never will. Perhaps you do not need to include the complexity of differentiating between yodeling dogs and barking dogs. This will make your system less complex, and it will provide the functionality that you need.

Deciding whether to design for less complexity or more functionality is a balancing act. The primary goal is always to build a system that is flexible without adding so much complexity that the system collapses under its own weight. What happens if you need to add yodeling at a later point in the project?

Current and future costs are also a major factor in these decisions. Although it might seem appropriate to make a system more complete and flexible, this added functionality might barely add any benefit—the return on investment might not be there. For example, would you extend the design of your `Dog` system to include other canines, such as hyenas and foxes (see Figure 7.5)?

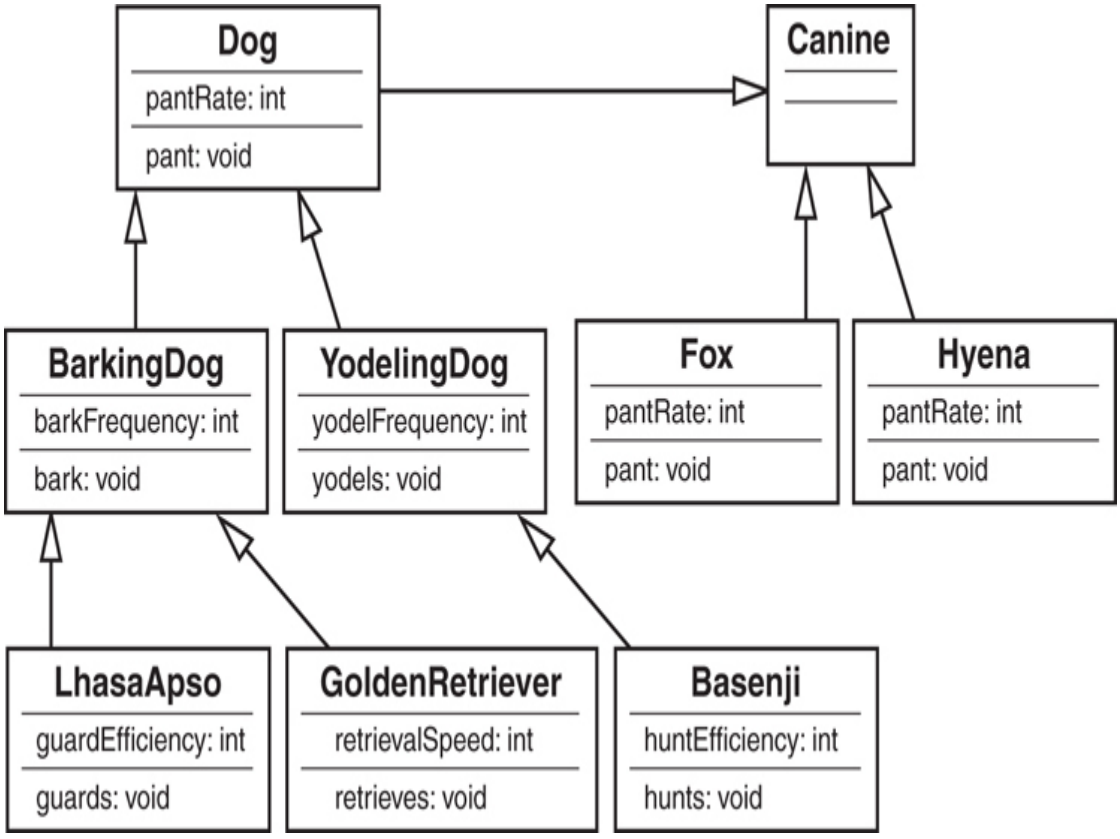

Figure 7.5 An expanded canine model.

Although this design might be prudent if you were a zookeeper, the extension of the `Canine` class is probably not necessary if you are breeding and selling domesticated dogs.

So as you can see, there are always trade-offs when creating a design.

> ### Making Design Decisions with the Future in Mind
>
> You might at this point say, "Never say never." Although you might not breed yodeling dogs now, sometime in the future you might want to do so. If you do not design for the possibility of yodeling dogs now, it will be much more expensive to change the system later to

> include them. This is yet another of the many design decisions that you have to make. You could possibly override the bark() method to make it yodel; however, this is not intuitive, and some people will expect a method called bark() to actually bark.

Composition

It is natural to think of objects as containing other objects. A television set contains a tuner and video display. A computer contains video cards, keyboards, and drives. The computer can be considered an object unto itself, and a flash drive is also considered a valid object. You could open up the computer and remove the hard drive and hold it in your hand. In fact, you could take the hard drive to another computer and install it. The fact that it is a standalone object is reinforced because it works in multiple computers.

The classic example of object composition is the automobile. Many books, training classes, and articles seem to use the automobile as the classic example of object composition. Besides the original interchangeable manufacture of the rifle, most people think of the automobile assembly line created by Henry Ford as the quintessential example of interchangeable parts. Thus, it seems natural that the automobile has become a primary reference point for designing OO software systems.

Most people would think it natural for a car to contain an engine. However, a car contains many objects besides an engine, including wheels, a steering wheel, and a stereo. Whenever a particular object is composed of other objects, and those objects are included as object fields, the new object is known as a *compound*, an *aggregate*, or a *composite object* (see Figure 7.6).

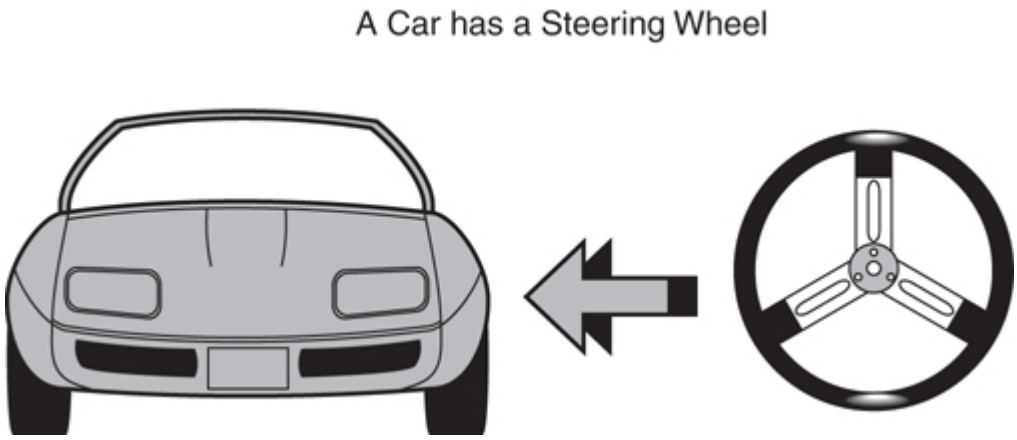

Figure 7.6 An example of composition.

> ### Aggregation, Association, and Composition
>
> From my perspective, there are only two ways to reuse classes—with inheritance or with composition. In Chapter 9, "Building Objects and Object-Oriented Design," we discuss composition in more detail—specifically, aggregation and association. In this book, I consider aggregation and association to be types of composition, although there are varied opinions on this.

Representing Composition with UML

To model the fact that the car object contains a steering wheel object, UML uses the notation shown in Figure 7.7.

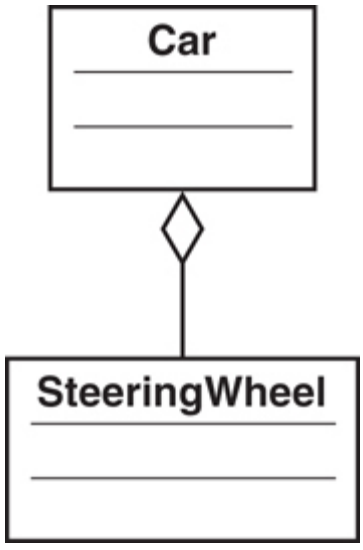

Figure 7.7 Representing composition in UML.

> ### Aggregation, Association, and UML
>
> In this book, aggregations are represented in UML by lines with a diamond, such as an engine as part of a car. Associations are represented by just the line (no diamond), such as a standalone keyboard servicing a separate computer box.

Note that the line connecting the `Car` class to the `SteeringWheel` class has a diamond shape on the `Car` side of the line. This signifies that a `Car` *contains* (has-a) `SteeringWheel`.

Let's expand this example. Suppose that none of the objects in this design use inheritance in any way. All the object relationships are strictly composition, and there are multiple levels of composition. Of course, this is a simplistic example, and there are many, many more object and object relationships in designing a car. However, this design is meant to be a simple illustration of what composition is all about.

Let's say that a car is composed of an engine, a stereo system, and a door.

> ### How Many Doors and Stereos?
>
> Note that a car normally has more than one door. Some have two, and some have four. You might even consider a hatchback a fifth door. In the same vein, it is not necessarily true that all cars have a stereo system. A car could have no stereo system or it could have one. I have even seen a car with two separate stereo systems. These situations are discussed in detail in Chapter 9. For the sake of this example, just pretend that a car has only a single door (perhaps it's a special racing car) and a single stereo system.

That a car is made up of an engine, a stereo system, and a door is easy to understand because most people think of cars in this way. However, it is important to keep in mind when designing software systems, just like automobiles, that objects are made up of other objects. In fact, the number of nodes and branches that can be included in this tree structure of classes is virtually unlimited.

Figure 7.8 shows the object model for the car, with the engine, stereo system, and door included.

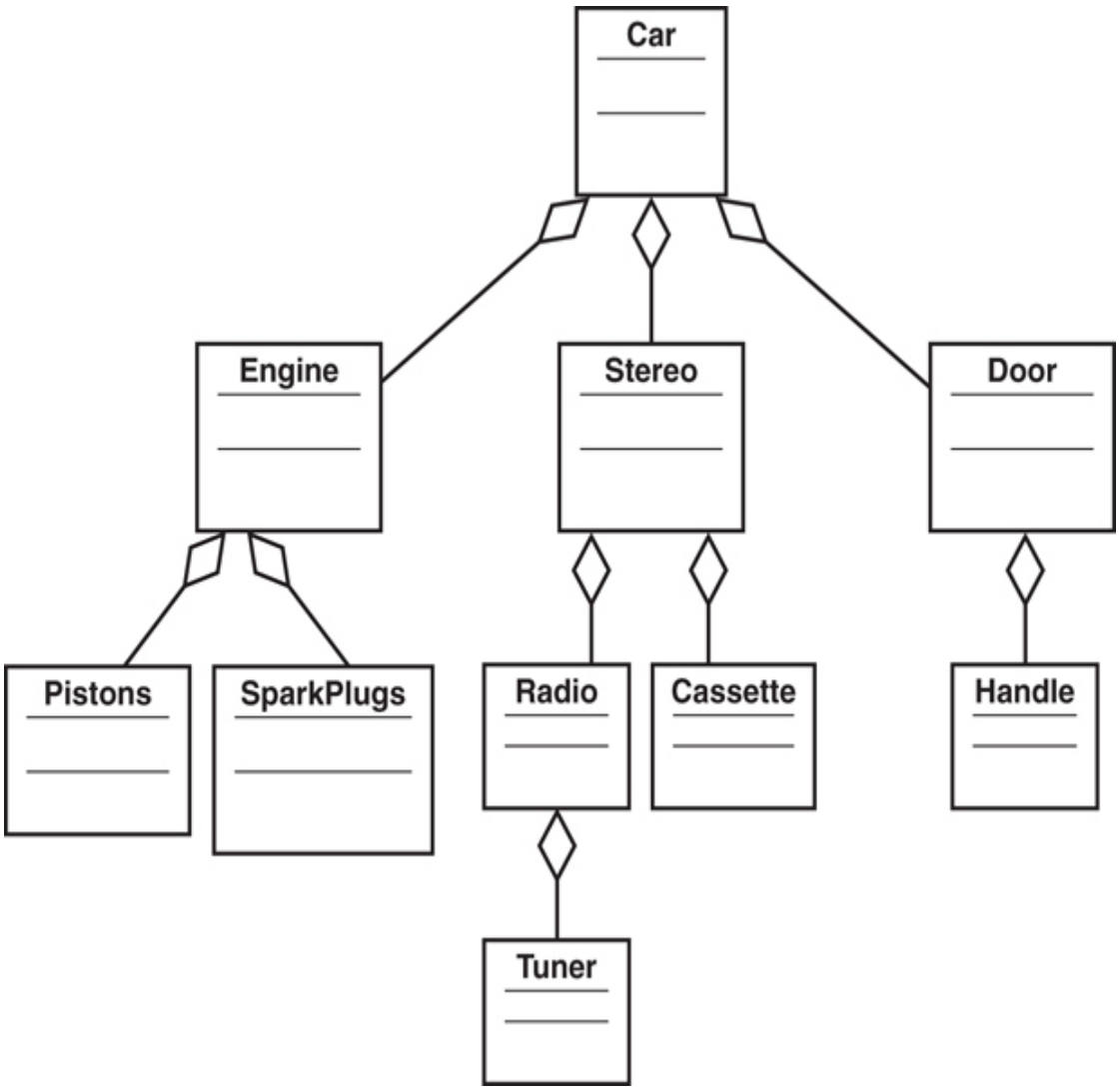

Figure 7.8 The `Car` class hierarchy.

Note that all three objects that make up a car are themselves composed of other objects. The engine contains pistons and spark plugs. The stereo contains a radio and a CD player. The door contains a handle. Also note that there is yet another level. The radio contains a tuner. We could have also added the fact that a handle contains a lock; the CD player contains a fast forward button, and so on. Additionally, we could have gone one level beyond the tuner and created an object for a dial. The level and complexity of the object model is up to the designer.

> **Model Complexity**
>
> As with the inheritance problem of the barking and yodeling dogs, using too much composition can also lead to more complexity. A fine line exists between creating an object model that contains enough granularity to be sufficiently expressive and a model that is so granular that it is difficult to understand and maintain.

Why Encapsulation Is Fundamental to OO

Encapsulation is the fundamental concept of OO. Whenever the interface/implementation paradigm is covered, we are talking about encapsulation. The basic question is what in a class should be exposed and what should not be exposed. This encapsulation pertains equally to data and behavior. When talking about a class, the primary design decision revolves around encapsulating both the data and the behavior into a well-written class.

Stephen Gilbert and Bill McCarty define encapsulation as "the process of packaging your program, dividing each of its classes into two distinct parts: the interface and the implementation." This is the message that has been presented over and over in this book.

But what does encapsulation have to do with inheritance, and how does it apply with regard to this chapter? This has to do with an OO paradox. Encapsulation is so fundamental to OO that it is one of OO design's cardinal rules. Inheritance is also considered one of the three primary OO concepts. However, in one way, inheritance actually breaks encapsulation! How can this be? Is it possible that two of the three primary concepts of OO are incompatible with each other? Let's explore this possibility.

How Inheritance Weakens Encapsulation

As already stated, encapsulation is the process of packaging classes into the public interface and the private implementation. In essence, a class hides everything that is not necessary for other classes to know about.

Peter Coad and Mark Mayfield make a case that when using inheritance, encapsulation is inherently weakened within a class hierarchy. They talk

about a specific risk: Inheritance connotes strong encapsulation with other classes but weak encapsulation between a superclass and its subclasses.

The problem is that if you inherit an implementation from a superclass and then change that implementation, the change from the superclass *ripples through* the class hierarchy. This rippling effect potentially affects all the subclasses. At first, this might not seem like a major problem; however, as we have seen, a rippling effect such as this can cause unanticipated problems. For example, testing can become a nightmare. In Chapter 6, "Designing with Objects," we talked about how encapsulation makes testing systems easier. In theory, if you create a class called `Cabbie` (see Figure 7.9) with the appropriate public interfaces, any change to the implementation of `Cabbie` should be transparent to all other classes. However, in any design a change to a superclass is certainly not transparent to a subclass. Do you see the conundrum?

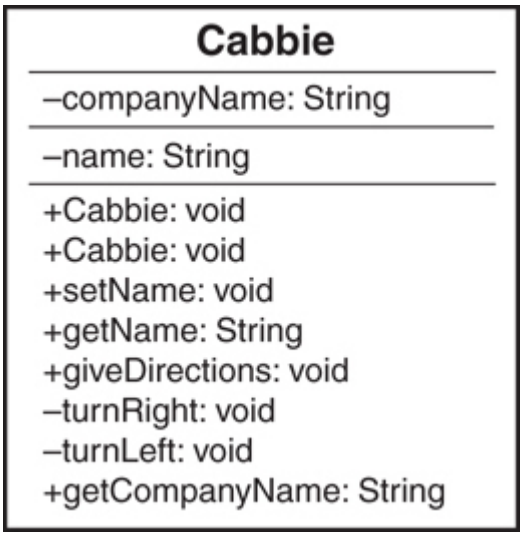

Figure 7.9 A UML diagram of the `Cabbie` class.

If the other classes were directly dependent on the implementation of the `Cabbie` class, testing would become more difficult, if not untenable. By using a different design approach, by abstracting out the behaviors and inheriting only attributes, these issues noted above go away.

If you then create a subclass of `Cabbie` called `PartTimeCabbie`, and `PartTimeCabbie` inherits the implementation from `Cabbie`, changing the implementation of `Cabbie` directly affects the `PartTimeCabbie` class.

For example, consider the UML diagram in Figure 7.10. `PartTimeCabbie` is a subclass of `Cabbie`. Thus, `PartTimeCabbie` inherits the public implementation of `Cabbie`, including the method `giveDirections()`. If the method `giveDirections()` is changed in `Cabbie`, it will have a direct impact on `PartTimeCabbie` and any other classes that might later be subclasses of `Cabbie`. In this subtle way, changes to the implementation of `Cabbie` are not necessarily encapsulated within the `Cabbie` class.

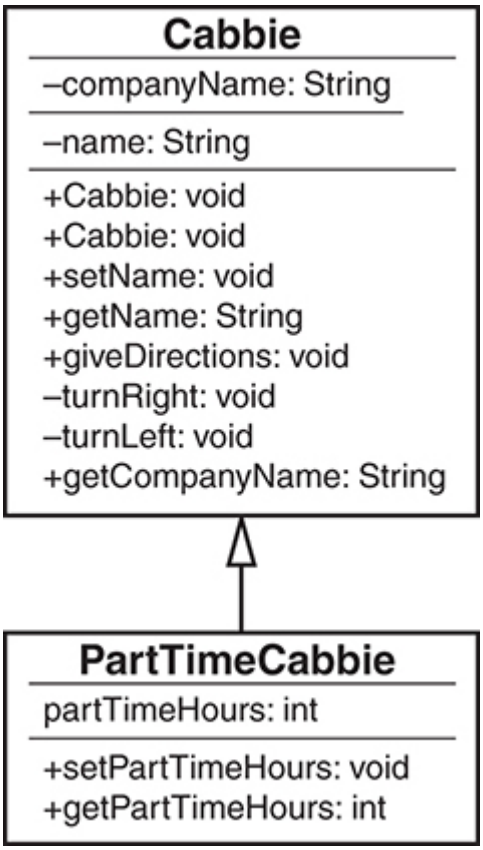

Figure 7.10 A UML diagram of the `Cabbie`/`PartTimeCabbie` classes.

To reduce the risk posed by this dilemma, it is important that you stick to the strict is-a condition when using inheritance. If the subclass were truly a specialization of the superclass, changes to the parent would likely affect the child in ways that are natural and expected. To illustrate, if a `Circle` class inherits implementation from a `Shape` class, and a change to the implementation of `Shape` breaks `Circle`, then `Circle` was not truly a `Shape` to begin with.

How can inheritance be used improperly? Consider a situation in which you want to create a window for the purposes of a graphical user interface (GUI). One impulse might be to create a window by making it a subclass of a rectangle class:

```
public class Rectangle {

}

public class Window extends Rectangle {

}
```

In reality a GUI window is much, much more than a rectangle. It is not a specialized version of a rectangle, as is a square. A true window might contain a rectangle (in fact, many rectangles); however, it is not a true rectangle. In this approach, a `Window` class should not inherit from `Rectangle`, but it should contain `Rectangle` classes.

```
public class Window {

    Rectangle menubar;
    Rectangle statusbar;
    Rectangle mainview;

}
```

A Detailed Example of Polymorphism

Many people consider polymorphism a cornerstone of OO design. Designing a class for the purpose of creating totally independent objects is what OO is all about. In a well-designed system, an object should be able to answer all the important questions about it. As a rule, an object should be responsible for itself. This independence is one of the primary mechanisms of code reuse.

As stated in Chapter 1, polymorphism literally means *many shapes*. When a message is sent to an object, the object must have a method defined to respond to that message. In an inheritance hierarchy, all subclasses inherit the interfaces from their superclass. However, because each subclass is a separate entity, each might require a separate response to the same message.

To review the example in Chapter 1, consider a class called `Shape`. This class has a behavior called `Draw`. However, when you tell somebody to draw a shape, the first question is likely to be, "What shape?" Simply telling a person to draw a shape is too abstract (in fact, the `Draw` method in `Shape` contains no implementation). You must specify which shape you mean. To do this, you provide the actual implementation in `Circle` and other subclasses. Even though `Shape` has a `Draw` method, `Circle` overrides this method and provides its own `Draw` method. Overriding basically means replacing an implementation of a parent with your own.

Object Responsibility

Let's revisit the `Shape` example from Chapter 1 (see Figure 7.11).

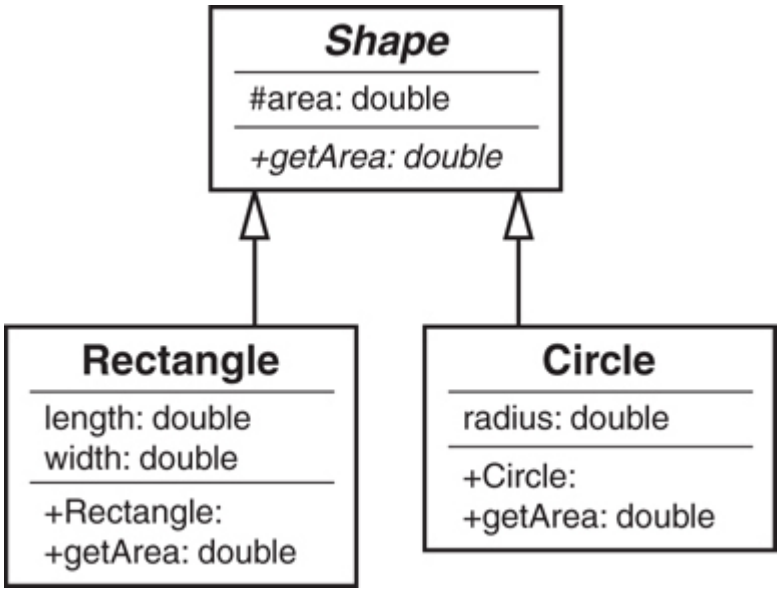

Figure 7.11 The `Shape` class hierarchy.

Polymorphism is one of the most elegant uses of inheritance. Remember that a `Shape` cannot be instantiated. It is an abstract class because it has an abstract method, `getArea()`. Chapter 8 explains abstract classes in great detail.

However, `Rectangle` and `Circle` can be instantiated because they are concrete classes. Although `Rectangle` and `Circle` are both shapes, they have some differences. As shapes, their area can be calculated. Yet the

formula to calculate the area is different for each. Thus, the area formulas cannot be placed in the `Shape` class.

This is where polymorphism comes in. The premise of polymorphism is that you can send messages to various objects, and they will respond according to their object's type. For example, if you send the message `getArea()` to a `Circle` class, you will invoke a different calculation than if you send the same `getArea()` message to a `Rectangle` class. This is because both `Circle` and `Rectangle` are responsible for themselves. If you ask `Circle` to return its area, it knows how to do this. If you want a circle to draw itself, it can do this as well. A `Shape` object could not do this even if it could be instantiated because it does not have enough information about itself. Notice that in the UML diagram (Figure 7.11), the `getArea()` method in the `Shape` class is italicized. This designates that the method is abstract.

As a very simple example, imagine that there are four classes: the abstract class `Shape`, and concrete classes `Circle`, `Rectangle`, and `Star`. Here is the code:

Click here to view code image

```
public abstract class Shape{

    public abstract void draw();

}
public class Circle extends Shape{

    public void draw() {

        System.out.println("I am drawing a Circle");

    }
}

public class Rectangle extends Shape{

    public void draw() {

        System.out.println("I am drawing a Rectangle");

    }
```

```
}

public class Star extends Shape{

    public void draw() {

        System.out.println("I am drawing a Star");

    }
}
```

Notice that only one method exists for each class: `draw()`. Here is the important point regarding polymorphism and an object being responsible for itself: The concrete classes themselves have responsibility for the drawing function. The `Shape` class does not provide the code for drawing; the `Circle`, `Rectangle`, and `Star` classes do this for themselves. Here is some code to prove it:

Click here to view code image

```
public class TestShape {

    public static void main(String args[]) {

        Circle circle = new Circle();
        Rectangle rectangle = new Rectangle();
        Star star = new Star();

        circle.draw();
        rectangle.draw();
        star.draw();

   }

}
```

The test application `TestShape` creates three classes: `Circle`, `Rectangle`, and `Star`. To draw these classes, `TestShape` asks the individual classes to draw themselves:

```
circle.draw();
rectangle.draw();
star.draw();
```

When you execute `TestShape`, you get the following results:

```
C:\>java TestShape
I am drawing a Circle
I am drawing a Rectangle
I am drawing a Star
```

This is polymorphism at work. What would happen if you wanted to create a new shape, such as `Triangle`? Simply write the class, compile it, test it, and use it. The base class `Shape` does not have to change—nor does any other code:

Click here to view code image

```
public class Triangle extends Shape{

    public void draw() {

        System.out.println("I am drawing a Triangle");

    }
}
```

A message can now be sent to `Triangle`. And even though `Shape` does not know how to draw a triangle, the `Triangle` class does:

Click here to view code image

```
public class TestShape {

    public static void main(String args[]) {

        Circle circle = new Circle();
        Rectangle rectangle = new Rectangle();
        Star star = new Star();
        Triangle triangle = new Triangle ();

        circle.draw();
        rectangle.draw();
        star.draw();
        triangle.draw();

   }

}

C:\>java TestShape
I am drawing a Circle
```

```
I am drawing a Rectangle
I am drawing a Star
I am drawing a Triangle
```

To see the real power of polymorphism, you can pass the shape to a method that has absolutely no idea what shape is coming. Take a look at the following code, which includes the specific shapes as parameters:

Click here to view code image

```
public class TestShape {

   public static void main(String args[]) {

     Circle circle = new Circle();
     Rectangle rectangle = new Rectangle();
     Star star = new Star();

     drawMe(circle);
     drawMe(rectangle);
     drawMe(star);

   }

   static void drawMe(Shape s) {
     s.draw();
   }

}
```

In this case, the `Shape` object can be passed to the method `drawMe()`, and the `drawMe()` method can handle any valid `Shape`—even one you add later. You can run this version of `TestShape` just like the previous one.

Abstract Classes, Virtual Methods, and Protocols

Abstract classes, as they are defined in Java, can be directly implemented in .NET and C++ as well. Not surprisingly, the C# .NET code looks similar to the Java code, as shown in the following:

Click here to view code image

```
public abstract class Shape{
```

```
   public abstract void draw();

}
```

The Visual Basic .NET code is written like this:

```
Public MustInherit Class Shape

    Public MustOverride Function draw()

End Class
```

The same functionality can be provided in C++ using virtual methods with the following code:

```
class Shape
{
    public:
        virtual void draw() = 0;
}
```

As mentioned in previous chapters, Objective-C and Swift do not fully implement the functionality of abstract classes.

For example, consider the following Java interface code for the `Shape` class we have seen many times:

```
public abstract class Shape{

    public abstract void draw();

}
```

The corresponding Objective-C (Swift) protocol is shown in the following code. Note that in both the Java code and the Objective-C code, there is no implementation for the `draw()` method.

```
@protocol Shape

@required
- (void) draw;

@end // Shape
```

At this point, the functionality for the abstract class and the protocol are pretty much equivalent; however, here is where the Java-type interface and protocols diverge. Consider the following Java code:

```
public abstract class Shape{
    public abstract void draw();
    public void print() {
        System.out.println("I am printing");
    };
}
```

In the preceding Java code, the `print` () method provides code that can be inherited by a subclass. Although this is also the case with C# .NET, VB .NET, and C++,the same cannot be said for an Objective-C protocol, which would look like this:

```
@protocol Shape

@required
- (void) draw;
- (void) print;

@end // Shape
```

In this protocol, the `print`() method signature is provided, and thus must be implemented by a subclass; however, no code can be included. In short, subclasses cannot directly inherit any code from a protocol. Thus, the protocol cannot be used in the same way as an abstract class, and this has implications when designing an object model.

Conclusion

This chapter gives a basic overview of what inheritance and composition are and how they are different. Many well-respected OO designers have stated that composition should be used whenever possible, and inheritance should be used only when necessary.

However, this is a bit simplistic. I believe that the idea that composition should be used whenever possible hides the real issue, which might be that composition is more appropriate in more cases than inheritance—not that it should be used whenever possible. The fact that composition might be more

appropriate in most cases does not mean that inheritance is evil. Use both composition and inheritance, but only in their proper contexts.

In earlier chapters, the concepts of abstract classes and Java interfaces arose several times. In Chapter 8, we explore the concept of development contracts and how abstract classes and Java interfaces are used to satisfy these contracts.

References

Booch, Grady and Robert A. Maksimchuk and Michael W. Engel and Bobbi J. Young, Jim Conallen, and Kelli A. Houston. 2007. *Object-Oriented Analysis and Design with Applications*, Third Edition. Boston, MA: Addison-Wesley.

Coad, Peter, and Mark Mayfield. 1997. *Java Design*. Upper Saddle River, NJ: Prentice Hall.

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

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

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

Chapter 7, "Mastering Inheritance and Composition," explains how inheritance and composition play major roles in the design of object-oriented (OO) systems. This chapter expands upon the concepts of interfaces, protocols, and abstract classes.

Interfaces, protocols, and abstract classes are powerful mechanisms for code reuse, providing the foundation for a concept I call *contracts*. This chapter covers the topics of code reuse, frameworks, contracts, interfaces, protocols, and abstract classes (for the remainder of the chapter, unless otherwise indicated, I use the term *interface* to include the concept of protocols). At the end of the chapter, we'll work through an example of how all these concepts can be applied to a real-world situation.

Code: To Reuse or Not to Reuse?

Programmers have been dealing with the issue of code reuse ever since writing their first line of code. Many software development paradigms stress code reuse as a major part of the process. Since the dawn of computer software, the concept of reusing code has been reinvented several times. The OO paradigm is no different. One of the major advantages touted by OO proponents is that if you write code properly the first time, you can reuse it to your heart's content.

This is true only to a certain degree. As with all design approaches, the utility and the reusability of code depend on how well it was designed and implemented. OO design does not hold the patent on code reuse. There is nothing stopping anyone from writing very robust and reusable code in a non-

OO language. Certainly, countless numbers of routines and functions, written in structured languages such as COBOL, C, and traditional VB, are of high quality and are quite reusable.

Thus, it is clear that following the OO paradigm is not the only way to develop reusable code. However, the OO approach does provide several mechanisms for facilitating the development of reusable code. One way to create reusable code is to create frameworks. In this chapter, we focus on using interfaces and abstract classes to create frameworks and encourage reusable code.

What Is a Framework?

Hand in hand with the concept of code reuse is the concept of *standardization*, which is sometimes called *plug and play*. The idea of a framework revolves around these plug-and-play and reuse principles. One classic example of a framework is a desktop application. Let's take an office suite application as an example. The document editor that I am currently using has a ribbon that includes multiple tab options. These options are similar to those in the presentation package and the spreadsheet software that I also have open. In fact, the first two menu items (Home, Insert) are the same in all three programs. Not only are the menu options similar, but many of the options look remarkably alike as well (New, Open, Save, and so on). Below the ribbon is the document area—whether it be for a document, a presentation, or a spreadsheet. The common framework makes it easier to learn various applications within the office suite. It also makes a developer's life easier by allowing maximum code reuse, not to mention that we can reuse portions of the design as well.

The fact that all these menu bars have a similar look and feel is obviously not an accident. In fact, when you develop in most integrated development environments, on a certain platform like Microsoft Windows or Linux, for example, you get certain things without having to create them yourself. When you create a window in a Windows environment, you get elements like the main title bar and the file Close button in the top-right corner. Actions are standardized as well—when you double-click the main title bar, the screen always minimizes/maximizes. When you click the Close button in the top-

right corner, the application always terminates. This is all part of the framework. Figure 8.1 is a screenshot of a word processor. Note the menu bars, toolbars, and other elements that are part of the framework.

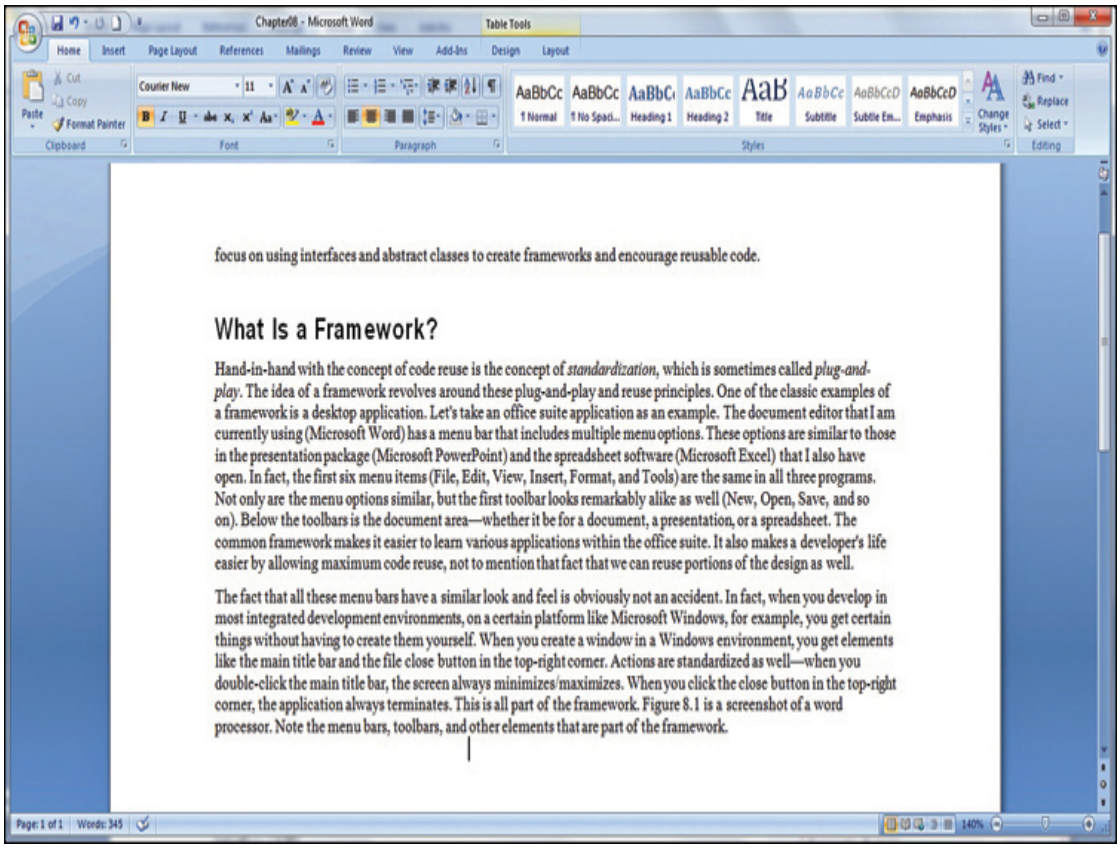

Figure 8.1 A word processing framework.

A word processing framework generally includes operations such as creating documents, opening documents, saving documents, cutting text, copying text, pasting text, searching through documents, and so on. To use this framework, a developer must use a predetermined interface to create an application. This predetermined interface conforms to the standard framework, which has two obvious advantages. First, as we have already seen, the look and feel are consistent, and the end users do not have to learn a new framework. Second, a developer can take advantage of code that has already been written and tested (and this testing issue is a huge advantage). Why write code to create a brand new Open dialog when one already exists and has been thoroughly tested? In a business setting, when time is critical, people do not want to have to learn new things unless it is absolutely necessary.

> ### Code Reuse Revisited
>
> In Chapter 7, we talked about code reuse as it pertains to inheritance—basically one class inheriting from another class. This chapter is about frameworks and reusing whole or partial systems.

The obvious question is this: If you need a dialog box, how do you use the dialog box provided by the framework? The answer is simple: Follow the rules that the framework provides. And where might you find these rules? The rules for the framework are found in the documentation. The person or persons who wrote the class, classes, or class libraries should have provided documentation on how to use the public interfaces of the class, classes, or class libraries (at least we hope). In many cases, this takes the form of the application-programming interface (API).

For example, to create a menu bar in Java, you would bring up the API documentation for the `JMenuBar` class and take a look at the public interfaces it presents. Figure 8.2 shows a part of the Java API. By using these APIs, you can create a valid Java application and conform to required standards. If you follow these standards, your application will be set to run in Java-enabled browsers.

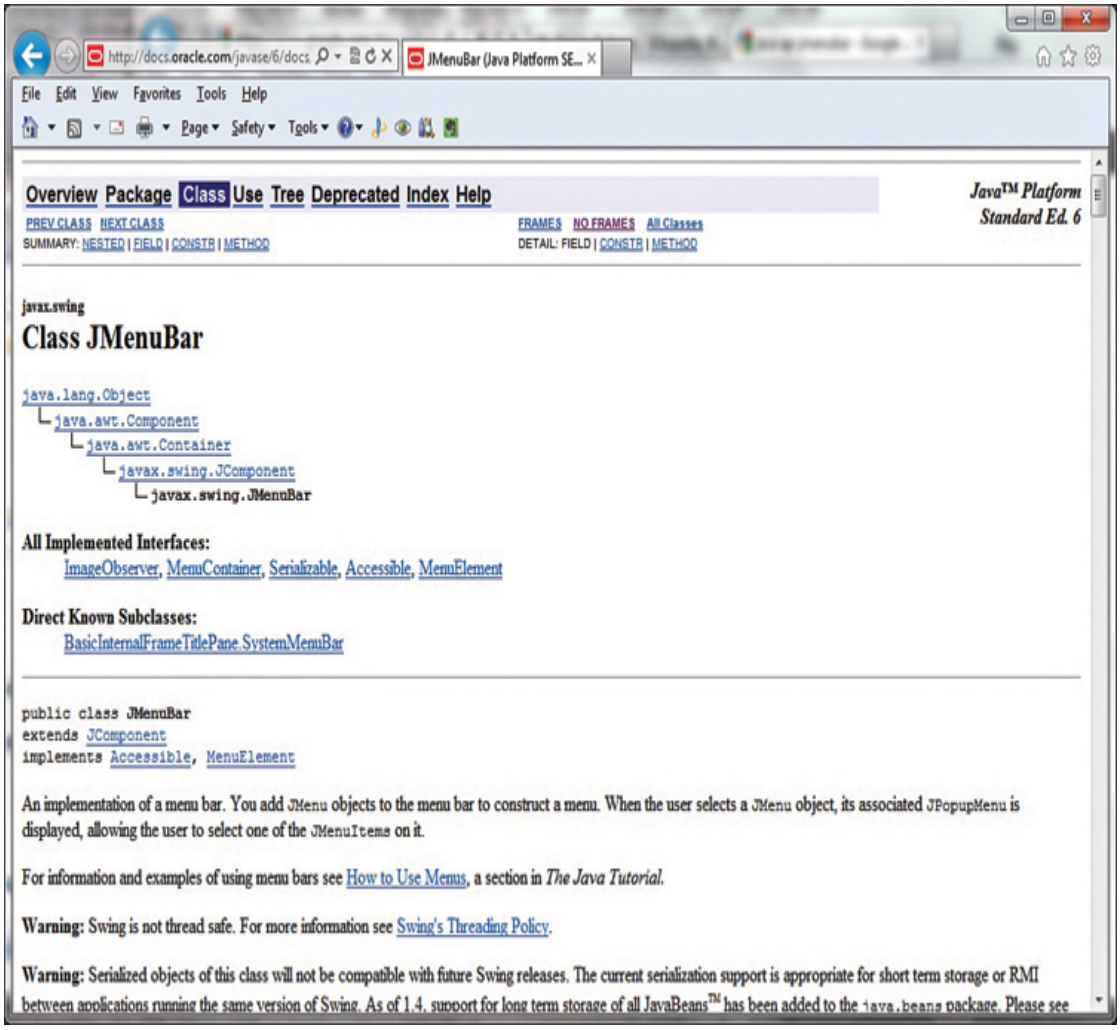

Figure 8.2 API documentation.

What Is a Contract?

In the context of this chapter, we will consider a *contract* to be any mechanism that requires a developer to comply with the specifications of an API. Often, an API is referred to as a framework. The online dictionary, Merriam-Webster (https://www.merriam-webster.com), defines a contract as a "binding agreement between two or more persons or parties, especially: one legally enforceable."

This is exactly what happens when a developer uses an API—with the project manager, business owner, or industry standard providing the enforcement. When using contracts, the developer is required to comply with the rules defined in the framework. This includes issues such as method

names, number of parameters, and so on (signatures, and the like). In short, standards are created to facilitate good development practices.

> ### The Term Contract
>
> The term contract is widely used in many aspects of business, including software development. Do not confuse the concept presented here with other possible software design concepts called contracts.

Enforcement is vital because it is always possible for a developer to break a contract. Without enforcement, a rogue developer could decide to reinvent the wheel and write her own code rather than use the specification provided by the framework. There is little benefit to a standard if people routinely disregard or circumvent it. In Java and the .NET languages, the two ways to implement contracts are to use abstract classes and interfaces.

Abstract Classes

One way a contract is implemented is via an abstract class. An *abstract class* is a class that contains one or more methods that do not have any implementation provided. Suppose that you have an abstract class called `Shape`. It is abstract because you cannot instantiate it. If you ask someone to draw a shape, the first thing the person will most likely ask you is, "What kind of shape?" Thus, the concept of a shape is abstract. However, if someone asks you to draw a circle, this does not pose quite the same problem, because a circle is a concrete concept. You know what a circle looks like. You also know how to draw other shapes, such as rectangles.

How does this apply to a contract? Let's assume that we want to create an application to draw shapes. Our goal is to draw every kind of shape represented in our current design, as well as ones that might be added later. There are two conditions we must adhere to.

First, we want all shapes to use the same syntax to draw themselves. For example, we want every shape implemented in our system to contain a method called `draw()`. Thus, seasoned developers implicitly know that to draw a shape, you invoke the `draw()` method, regardless of what the shape

happens to be. Theoretically, this reduces the amount of time spent fumbling through manuals, and it cuts down on syntax errors.

Second, remember that it is important that every class be responsible for its own actions. Thus, even though a class is required to provide a method called `draw()`, that class must provide its own implementation of the code. For example, the classes `Circle` and `Rectangle` both have a `draw()` method; however, the `Circle` class obviously has code to draw a circle, and as expected, the `Rectangle` class has code to draw a rectangle. When we ultimately create classes called `Circle` and `Rectangle`, which are subclasses of `Shape`, these classes must implement their own version of `draw()` (see Figure 8.3).

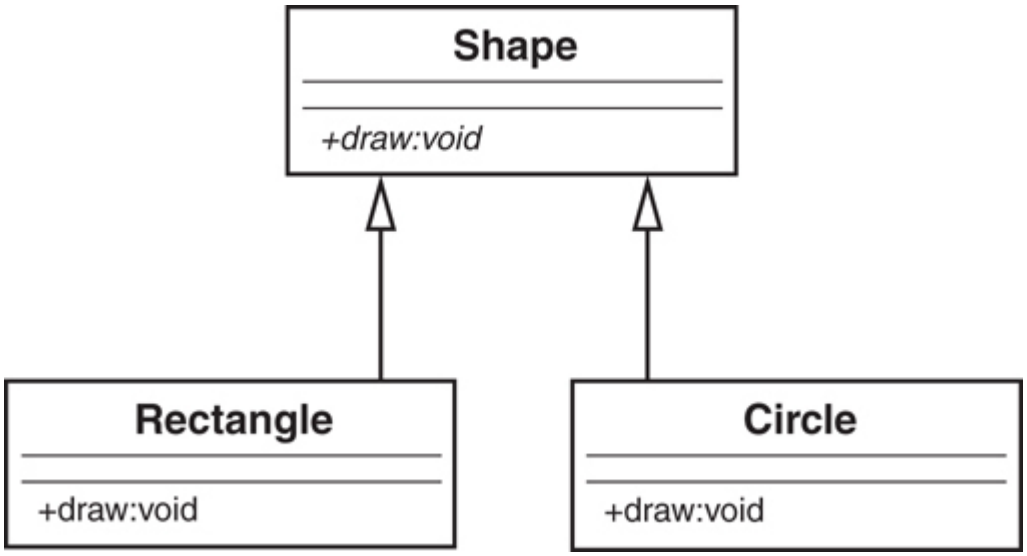

Figure 8.3 An abstract class hierarchy.

> ### Abstract Methods
>
> In the UML diagrams, note that the abstract methods are italicized.

In this way, we have a `Shape` framework that is truly polymorphic. The `draw()` method can be invoked for every single shape in the system, and invoking each shape produces a different result. Invoking the `draw()` method on a `Circle` object draws a circle, and invoking the `draw()` method on a `Rectangle` object draws a rectangle. In essence, sending a message to an object evokes a different response, depending on the object. This is the essence of polymorphism.

```
circle.draw();          // draws a circle
rectangle.draw();     // draws a rectangle
```

Let's look at some code to illustrate how `Rectangle` and `Circle` conform to the `Shape` contract. Here is the code for the `Shape` class:

```
public abstract class Shape {

    public abstract void draw(); // no implementation

}
```

Note that the class does not provide any implementation for `draw();` basically there is no code, and this is what makes the method abstract (providing any code would make the method concrete). There are two reasons why there is no implementation. First, `Shape` does not know what to draw, so we could not implement the `draw()` method even if we wanted to.

> ### Structured Analogy
>
> This is an interesting issue. If we did want the Shape class to contain the code for all possible shapes, present and future, conditional statements, such as a Java switch statement, would be required. This would be very messy and difficult to maintain. This is one example of where the strength of an object-oriented design comes into play.

Second, we want the subclasses to provide the implementation. Let's look at the `Circle` and `Rectangle` classes:

Click here to view code image

```
public class Circle extends Shape {

    public void Draw() {System.out.println ("Draw a
Circle")};

}

public class Rectangle extends Shape {

    public void Draw() {System.out.println ("Draw a
Rectangle")};
```

```
}
```

Note that both `Circle` and `Rectangle` extend (that is, inherit from) `Shape`. Also notice that they provide the actual implementation (in this case, the implementation is trivial). Here is where the contract comes in. If `Circle` inherits from `Shape` and fails to provide a `draw()` method, `Circle` won't even compile. Thus, `Circle` would fail to satisfy the contract with `Shape`. A project manager can require that programmers creating shapes for the application must inherit from `Shape`. By doing this, all shapes in the application will have a `draw()` method that performs in an expected manner.

> ### Circle
>
> If `Circle` does indeed fail to implement a `draw()` method, `Circle` will be considered abstract itself. Thus, yet another subclass must inherit from `Circle` and implement a `draw()` method. This subclass would then become the concrete implementation of both Shape and `Circle`.

Although the concept of abstract classes revolves around abstract methods, nothing is stopping `Shape` from providing some implementation. Remember that the definition for an abstract class is that it contains *one or more* abstract methods—this implies that an abstract class can also provide concrete methods. For example, although `Circle` and `Rectangle` implement the `draw()` method differently, they share the same mechanism for setting the color of the shape. So, the `Shape` class can have a color attribute and a method to set the color. This `setColor()` method is a concrete implementation and would be inherited by both `Circle` and `Rectangle`. The only methods that a subclass must implement are the ones that the superclass declares as abstract. These abstract methods are the contract.

> ### Caution
>
> Be aware that in the cases of `Shape`, `Circle`, and `Rectangle`, we are dealing with a strict inheritance relationship, as opposed to an interface, which we discuss in the next section. `Circle` *is-a* Shape, and `Rectangle` *is-a* `Shape`.

Some languages, such as C++, use only abstract classes to implement contracts; however, Java and .NET have another mechanism that implements a contract called an interface. In other cases, such as Objective-C and Swift, abstract classes are not provided by the language. Thus, to implement a contract in Objective-C or Swift, you need to use protocols.

Interfaces

Before defining an interface, it is interesting to note that C++ does not have a construct called an interface. When using C++, you can essentially create an interface by using a syntax subset of an abstract class. For example, the following C++ code is an abstract class. However, because the only method in the class is a virtual method, there is no implementation. As a result, this abstract class provides the same functionality as an interface.

```
class Shape
{
    public:
        virtual void draw() = 0;
}
```

> ### Interface Terminology
>
> This is another one of those times when software terminology gets confusing—very confusing. Be aware that you can use the term interface in several ways, so be sure to use each in the proper context.
>
> First, the graphical user interface (GUI) is widely used when referring to the visual interface that a user interacts with—often on a monitor.
>
> Second, the interface to a class is basically the signatures of its methods.
>
> Third, in Objective-C and Swift, you break up the code into physically separate modules called the interface and implementation.

> Fourth, an interface and a protocol are basically a contract between a parent class and a child class.Can you think of any others?

The obvious question is this: If an abstract class can provide the same functionality as an interface, why do Java and .NET bother to provide this construct called an interface? And why does Objective-C and Swift provide the protocol?

For one thing, C++ supports multiple inheritance, whereas Java, Objective-C, Swift, and .NET do not. Although Java, Objective-C, Swift, and .NET classes can inherit from only one parent class, they can implement many interfaces. Using more than one abstract class constitutes multiple inheritance; thus, Java and .NET cannot go this route. In short, when using an interface, you do not have to concern yourself with a formal inheritance structure—you can theoretically add an interface to any class if the design makes sense. However, an abstract class requires you to inherit from that abstract class and, by extension, all of its potential parents.

> **Circle**
>
> Because of these considerations, interfaces are often thought to be a workaround for the lack of multiple inheritance. This is not technically true. Interfaces are a separate design technique, and although they can be used to design applications that could be done with multiple inheri-tance, they do not replace or circumvent multiple inheritance.

As with abstract classes, interfaces are a powerful way to enforce contracts for a framework. Before we get into any conceptual definitions, it's helpful to see an actual interface UML diagram and the corresponding code. Consider an interface called `Nameable`, as shown in Figure 8.4.

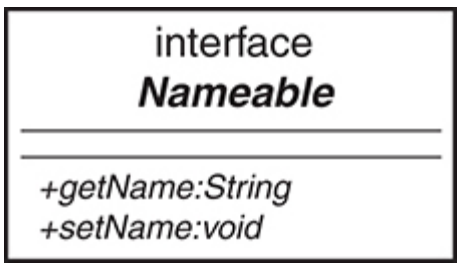

Figure 8.4 A UML diagram of a Java interface.

Note that `Nameable` is identified in the UML diagram as an interface, which distinguishes it from a regular class (abstract or not). Also note that the interface contains two methods, `getName()` and `setName()`. Here is the corresponding code:

```
public interface Nameable {

    String getName();
    void setName (String aName);

}
```

For comparison purposes, here is the code for the corresponding Objective-C protocol:

```
@protocol Nameable

@required
- (char *) getName;
- (void) setName: (char *) n;
@end // Nameable
```

In the code, notice that `Nameable` is not declared as a class but as an interface. Because of this, both methods, `getName()` and `setName()`, are considered abstract and no implementation is provided. An interface, unlike an abstract class, can provide *no* implementation at all. As a result, any class that implements an interface must provide the implementation for all methods. For example, in Java, a class inherits from an abstract class, whereas a class implements an interface.

> ### Implementation Versus Definition Inheritance
>
> Sometimes inheritance is referred to as implementation inheritance, and interfaces are called *definition inheritance*.

Tying It All Together

If both abstract classes and interfaces provide abstract methods, what is the real difference between the two? As we saw before, an abstract class can provides both abstract and concrete methods, whereas an interface provides only abstract methods. Why is there such a difference?

Assume that we want to design a class that represents a dog, with the intent of adding more mammals later. The logical move would be to create an abstract class called `Mammal`:

Click here to view code image

```
public abstract class Mammal {

    public void generateHeat() {System.out.println("Generate
heat");}

    public abstract void makeNoise();

}
```

This class has a concrete method called `generateHeat()` and an abstract method called `makeNoise()`. The method `generateHeat()` is concrete because all mammals generate heat. The method `makeNoise()` is abstract because each mammal will make noise differently.

Let's also create a class called `Head` that we will use in a composition relationship:

Click here to view code image

```
public class Head {

    String size;

    public String getSize() {

        return size;

    }

    public void setSize(String aSize) { size = aSize; }

}
```

`Head` has two methods: `getSize()` and `setSize()`. Although composition might not shed much light on the difference between abstract classes and interfaces, using composition in this example does illustrate how composition relates to abstract classes and interfaces in the overall design of

an object-oriented system. I feel that this is important because the example is more complete. Remember that there are two ways to build object relationships: the *is-a* relationship, represented by inheritance, and the *has-a* relationship, represented by composition. The question is: Where does the interface fit in?

To answer this question and tie everything together, let's create a class called `Dog` that is a subclass of `Mammal`, implements `Nameable`, and has a `Head` object (see Figure 8.5).

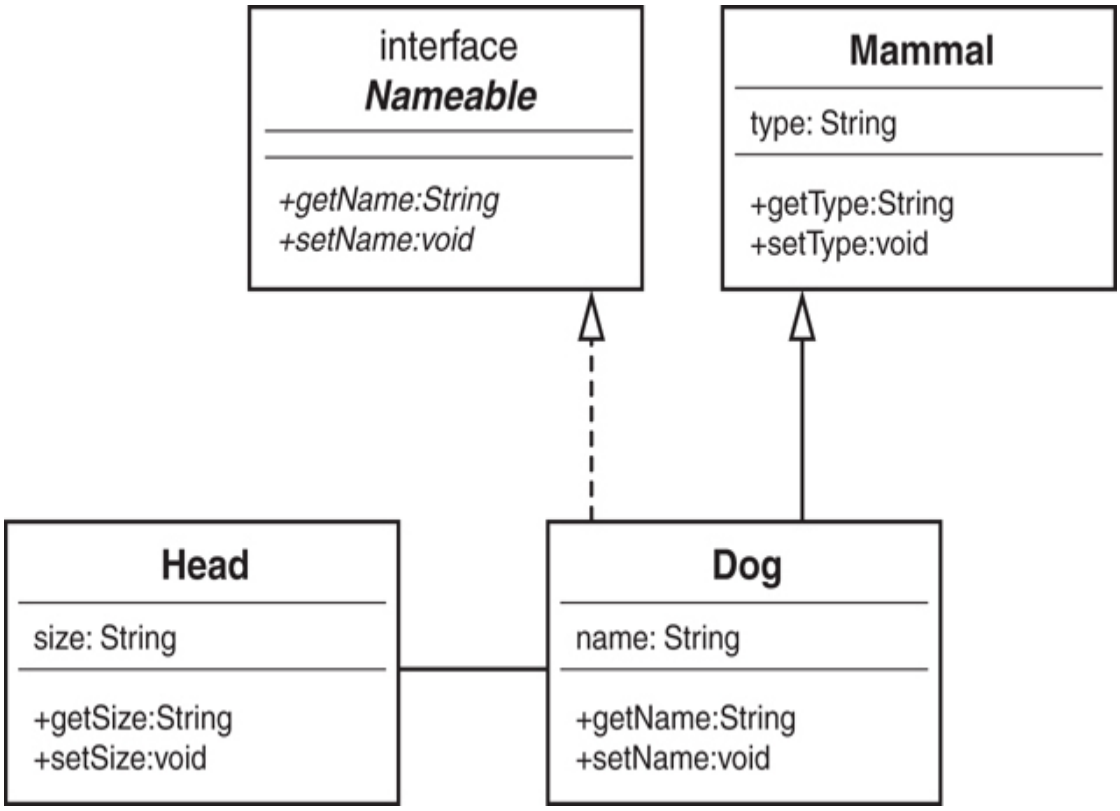

Figure 8.5 A UML diagram of the sample code.

In a nutshell, Java and .NET build objects in three ways: inheritance, interfaces, and composition. Note the dashed line in Figure 8.5 that represents the interface. This example illustrates when you should use each of these constructs. When do you choose an abstract class? When do you choose an interface? When do you choose composition? Let's explore further.

You should be familiar with the following concepts:

- `Dog` is a `Mammal`, so the relationship is inheritance.

- `Dog` implements `Nameable`, so the relationship is an interface.
- `Dog` has a `Head`, so the relationship is composition.

The following code shows how you would incorporate an abstract class and an interface in the same class:

Click here to view code image

```
public class Dog extends Mammal implements Nameable {

    String name;

    Head head;

    public void makeNoise(){System.out.println("Bark");}

    public void setName (String aName) {name = aName;}
    public String getName () {return (name);}

}
```

After looking at the UML diagram, you might come up with an obvious question: Even though the dashed line from `Dog` to `Nameable` represents an interface, isn't it still inheritance? At first glance, the answer is not simple. Although interfaces may well be considered a special type of inheritance, it is important to know what *special* means. Understanding these *special* differences is key to understanding a solid object-oriented design.

Although inheritance is a strict is-a relationship, an interface is not quite. For example:

- A dog is a mammal.
- A reptile is not a mammal.

Thus, a `Reptile` class could not inherit from the `Mammal` class. However, an interface transcends the various classes. For example:

- A dog is nameable.
- A lizard is nameable.

The key here is that classes in a strict inheritance relationship must be related. For example, in this design, the `Dog` class is directly related to the `Mammal` class. A dog is a mammal. Dogs and lizards are not related at the mammal level because you can't say that a lizard is a mammal. However, interfaces can be used for classes that are not related. You can name a dog just as well as you can name a lizard. This is the key difference between using an abstract class and using an interface.

The abstract class represents some sort of implementation. In fact, we saw that `Mammal` provided a concrete method called `generateHeat()`. Even though we do not know what kind of mammal we have, we know that all mammals generate heat. However, an interface models only behavior. An interface *never* provides any type of implementation, only behavior. The interface specifies behavior that is the same across classes that conceivably have no connection. Not only are dogs nameable, but so are cars, planets, and so on.

Some say that interfaces are a poor substitute for multiple inheritance. While it may be true that interfaces were part of the same Java design that eliminated multiple inheritance (and were adopted by many other languages), interfaces are used in different design situations than inheritance, as the `Nameable` example illustrates.

The Compiler Proof

Can we prove or disprove that interfaces have a true is-a relationship? In the case of Java (and this can also be done in C# or VB), we can let the compiler tell us. Consider the following code:

```
Dog D = new Dog();
Head H = D;
```

When this code is run through the compiler, the following error is produced:

Click here to view code image

```
Test.java:6: Incompatible type for Identifier. Can't convert
Dog to Head. Head H = D;
```

Obviously, a dog is not a head. Not only do we know this, but the compiler agrees. However, as expected, the following code works just fine:

```
Dog D = new Dog();
Mammal M = D;
```

This is a true inheritance relationship, and it is not surprising that the compiler parses this code cleanly because a dog is a mammal.

Now we can perform the true test of the interface. Is an interface an actual is-a relationship? The compiler thinks so:

```
Dog D = new Dog();
Nameable N = D;
```

This code works fine. So, we can safely say that a dog is a nameable entity. This is a simple but effective proof that both inheritance and interfaces constitute an is-a relationship. The interface relationship is more like a "behaves-like-a" when used properly. You might have data interfaces that are "is-a," but more often you are going to have the former.

> **Nameable Interface**
>
> An interface specifies certain behavior but not the implementation. By implementing the `Nameable` interface, you are saying that you will provide nameable behavior by implementing methods called `getName()` and `setName()`. How you implement these methods is up to you. All you have to do is to provide the methods.

Making a Contract

The simple rule for defining a contract is to provide an unimplemented method, via either an abstract class or an interface. Thus, when a subclass is designed with the intent of complying with the contract, it must provide the implementation for the unimplemented methods in the parent class or interface.

As stated earlier, one of the advantages of a contract is to standardize coding conventions. Let's explore this concept in greater detail by providing an example of what happens when coding standards are not used. In this case,

there are three classes: `Planet`, `Car`, and `Dog`. Each class implements code to name the entity. However, because they are all implemented separately, each class has different syntax to retrieve the name. Consider the following code for the `Planet` class:

Click here to view code image

```
public class Planet {
    String planetName;
    public void getPlanetName() {return planetName;};
}
```

Likewise, the `Car` class might have code like this:

Click here to view code image

```
public class Car {

    String carName;

    public String getCarName() { return carName; };

}
```

And the `Dog` class might have code like this:

```
public class Dog {

    String dogName;

    public String getDogName() { return dogName; };

}
```

The obvious issue here is that anyone using these classes would have to look at the documentation (what a horrible thought!) to figure out how to retrieve the name in each of these cases. Even though looking at the documentation is not the worst fate in the world, it would be nice if all the classes used in a project (or company) would use the same naming convention—it would make life a bit easier. This is where the `Nameable` interface comes in.

The idea would be to make a contract for any type of class that needs to use a name. As users of various classes move from one class to the other, they

would not have to figure out the current syntax for naming an object. The `Planet` class, the `Car` class, and the `Dog` class would all have the same naming syntax.

To implement this lofty goal, we can create an interface (we can use the `Nameable` interface that we used previously). The convention is that all classes must implement `Nameable`. In this way, the users have to remember only a single interface for all classes when it comes to naming conventions:

Click here to view code image

```
public interface Nameable {

    public String getName();
    public void setName(String aName);

}
```

The new classes, `Planet`, `Car`, and `Dog`, should look like this:

```
public class Planet implements Nameable {

    String planetName;

    public String getName() {return planetName;}
    public void setName(String myName) { planetName = myName;
}

}

public class Car implements Nameable {

    String carName;

    public String getName() {return carName;}
    public void setName(String myName) { carName = myName;}

}

public class Dog implements Nameable {

    String dogName;

    public String getName() {return dogName;}
    public void setName(String myName) { dogName = myName;}
```

```
}
```

In this way, we have a standard interface, and we've used a contract to ensure that it is the case. In fact, one of the major benefits of using a modern IDE is that, when implementing an interface, the IDE will automatically stub out the required methods. This feature saves lots of time and effort when using interfaces.

There is one little issue that you might have thought about. The idea of a contract is great as long as everyone plays by the rules, but what if some shady individual doesn't want to play by the rules (the rogue programmer)? The bottom line is that there is nothing to stop people from breaking the standard contract; however, in some cases, doing so will get them in deep trouble.

On one level, a project manager can insist that everyone use the contract, just like team members must use the same variable naming conventions and configuration management system. If a team member fails to abide by the rules, he could be reprimanded, or even fired.

Enforcing rules is one way to ensure that contracts are followed, but there are instances in which breaking a contract will result in unusable code. Consider the Java interface `Runnable`. Old-style Java applets implement the `Runnable` interface because it requires that any class implementing `Runnable` must implement a `run()` method. This is important because the browser that calls the applet will call the `run()` method within `Runnable`. If the `run()` method does not exist, things will break.

System Plug-in Points

Basically, contracts are "plug-in points" into your code. Anyplace where you want to make parts of a system abstract, you can use a contract. Instead of coupling to objects of specific classes, you can connect to any object that implements the contract. You need to be aware of where contracts are useful; however, you can overuse them. You want to identify common features such as the `Nameable` interface, as discussed in this chapter. However, be aware that there is a trade-off when using contracts. They might make code reuse more of a reality, but they make things somewhat more complex.

An E-Business Example

It's sometimes hard to convince a decision maker, who may have no development background, of the monetary savings of code reuse. However, when reusing code, it is pretty easy to understand the advantage to the bottom line. In this section, we'll walk through a simple but practical example of how to create a workable framework using inheritance, abstract classes, interfaces, and composition.

An E-Business Problem

Perhaps the best way to understand the power of reuse is to present an example of how you would reuse code. In this example, we'll use inheritance (via interfaces and abstract classes) and composition. Our goal is to create a framework that will make code reuse a reality, reduce coding time, and reduce maintenance—all the typical software development wish-list items.

Let's start our own Internet business. Let's assume that we have a client, a small pizza shop called Papa's Pizza. Despite the fact that it is a small, family-owned business, Papa realizes that a Web presence can help the business in many ways. Papa wants his customers to access his website, find out what Papa's Pizza is all about, and order pizzas right from the comfort of their browsers.

At the site we develop, customers will be able to access the website, select the products they want to order, and select a delivery option and time for delivery. They can eat their food at the restaurant, pick up the order, or have the order delivered. For example, a customer decides at 3:00 that he wants to order a pizza dinner (with salads, breadsticks, and drinks), to be delivered to his home at 6:00. Let's say the customer is at work (on a break, of course). He gets on the Web and selects the pizzas, including size, toppings, and crust; the salads, including dressings; breadsticks; and drinks. He chooses the delivery option and requests that the food be delivered to his home at 6:00. Then he pays for the order by credit card, gets a confirmation number, and exits. Within a few minutes he gets an email confirmation as well. We will set up accounts so that when people bring up the site, they will get a greeting reminding them of who they are, what their favorite pizza is, and what new pizzas have been created this week.

When the software system is finally delivered, it is deemed a total success. For the next several weeks, Papa's customers happily order pizzas and other food and drinks over the Internet. During this rollout period, Papa's brother-in-law, who owns a donut shop called Dad's Donuts, pays Papa a visit. Papa shows Dad the system, and Dad falls in love with it. The next day, Dad calls our company and asks us to develop a Web-based system for his donut shop. This is great, and exactly what we had hoped for. Now, how can we leverage the code that we used for the pizza shop in the system for the donut shop?

How many more small businesses, besides Papa's Pizza and Dad's Donuts, could take advantage of our framework to get on the Web? If we can develop a good, solid framework, we will be able to efficiently deliver Web-based systems at lower costs than we were able to do before. There will also be an added advantage that the code will have been tested and implemented previously, so debugging and maintenance should be greatly reduced.

The Non-Reuse Approach

For many reasons, the concept of code reuse has not been as successful as some software developers would like. First, many times reuse is not even considered when developing a system. Second, even when reuse is entered into the equation, the issues of schedule constraints, limited resources, and budgetary concerns often short-circuit the best intentions.

In many instances, code ends up highly coupled to the specific application for which it was written. This means that the code within the application is highly dependent on other code within the same application.

A lot of code reuse is the result of using cut, copy, and paste operations. While one application is open in a text editor, you copy code and then paste it into another application. Sometimes certain functions or routines can be used without any change. As is unfortunately often the case, even though most of the code may remain identical, a small bit of code must change to work in a specific application.

For example, consider two separate applications, as represented by the UML diagram in Figure 8.6.

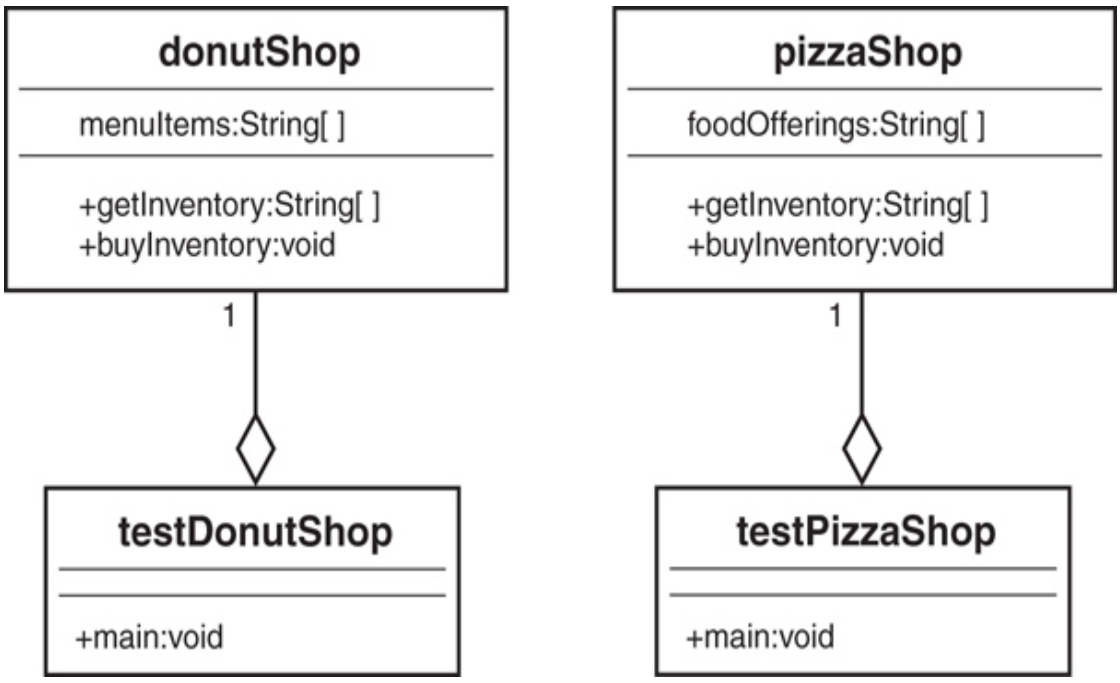

Figure 8.6 Applications on divergent paths.

In this example, the applications `testDonutShop` and `testPizzaShop` are totally independent code modules. The code is kept separate, and there is no interaction between the modules. However, these applications might use some common code. In fact, some code might have been copied verbatim from one application to another. At some point, someone involved with the project might decide to create a library of these shared pieces of code to use in these and other applications. In many well-run and disciplined projects, this approach works well. Coding standards, configuration management, change management, and so on are all very well run. However, in many instances, this discipline breaks down.

Anyone who is familiar with the software development process knows that when bugs crop up and time is of the essence, there is the temptation to put some fixes or additions into a system that are specific to the application currently in distress. This might fix the problem for the distressed application but could have unintended, possibly harmful, implications for other applications. Thus, in situations like these, the initially shared code can diverge, and separate code bases must be maintained.

For example, one day Papa's website crashes. He calls us in a panic, and one of our developers is able to track down the problem. The developer

fixes the problem, knowing that the fix works but is not quite sure why. The developer also does not know what other areas of the system the fix might inadvertently affect. So the developer makes a copy of the code, strictly for use in the Papa's Pizza system. This is affectionately named Version 2.01papa. Because the developer does not yet totally understand the problem and because Dad's system is working fine, the code is not migrated to the donut shop's system.

> ### Tracking Down a Bug
>
> The fact that the bug turned up in the pizza system does not mean that it will also turn up in the donut system. Even though the bug caused a crash in the pizza shop, the donut shop might never encounter it. It may be that the fix to the pizza shop's code is more dangerous to the donut shop than the original bug.

The next week Dad calls in a panic, with a totally unrelated problem. A developer fixes it, again not knowing how the fix will affect the rest of the system, makes a separate copy of the code, and calls it Version 2.03dad. This scenario gets played out for all the sites we now have in operation. There are now a dozen or more copies of the code, with various versions for the various sites. This becomes a mess. We have multiple code paths and have crossed the point of no return. We can never merge them again. (Perhaps we could, but from a business perspective, this would be costly.)

Our goal is to avoid the mess of the previous example. Although many systems must deal with legacy issues, fortunately for us, the pizza and donut applications are brand-new systems. Thus, we can use a bit of foresight and design this system in a reusable manner. In this way, we will not run into the maintenance nightmare just described. What we want to do is factor out as much commonality as possible. In our design, we will focus on all the common business functions that exist in a Web-based application. Instead of having multiple application classes like `testPizzaShop` and `testDonutShop`, we can create a design that has a class called `Shop` that all the applications will use.

Notice that `testPizzaShop` and `testDonutShop` have similar interfaces, `getInventory()` and `buyInventory()`. We will factor

out this commonality and require that all applications that conform to our `Shop` framework implement `getInventory()` and `buyInventory()` methods. This requirement to conform to a standard is sometimes called a contract. By explicitly setting forth a contract of services, you isolate the code from a single implementation. In Java, you can implement a contract by using an interface or an abstract class. Let's explore how this is accomplished.

An E-Business Solution

Now let's show how to use a contract to factor out some of the commonality of these systems. In this case, we will create an abstract class to factor out some of the implementation, and an interface (our familiar `Nameable`) to factor out some behavior.

Our goal is to provide customized versions of our Web application with the following features:

- An interface, called `Nameable`, which is part of the contract.
- An abstract class, called `Shop`, which is also part of the contract.
- A class called `CustList`, which we use in composition.
- A new implementation of `Shop` for each customer we service.

The UML Object Model

The newly created `Shop` class is where the functionality is factored out. Notice in Figure 8.7 that the methods `getInventory()` and `buyInventory()` have been moved up the hierarchy tree from `DonutShop` and `PizzaShop` to the abstract class `Shop`. Now, whenever we want to provide a new, customized version of `Shop`, we plug in a new implementation of `Shop` (such as a grocery shop). `Shop` is the contract that the implementations must abide by.

Click here to view code image

```
public abstract class Shop  {

    CustList customerList;
```

```
    public void CalculateSaleTax() {
        System.out.println("Calculate Sales Tax");
    }
    public abstract String[] getInventory();
    public abstract void buyInventory(String item);
}
```

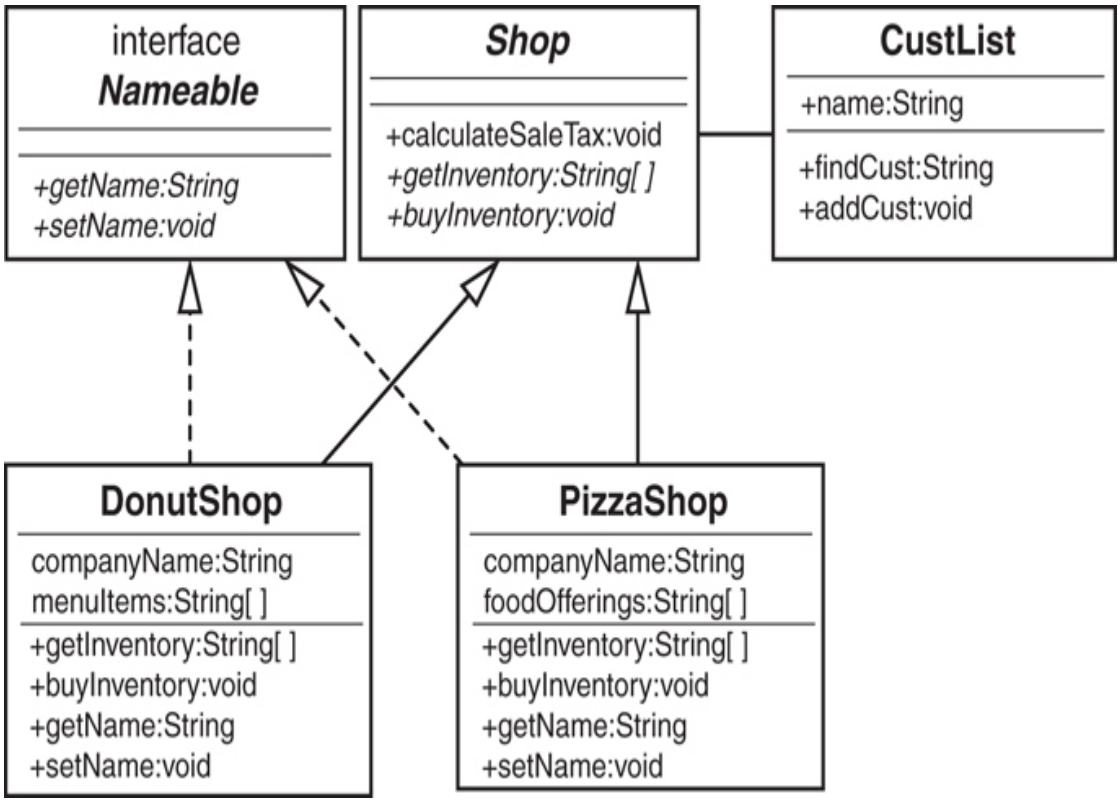

Figure 8.7 A UML diagram of the `Shop` model.

To show how composition fits into this picture, the `Shop` class has a customer list. Thus, the class `CustList` is contained within `Shop`:

Click here to view code image

```
public class CustList {
    String name;
```

```
    public  String findCust() {return name;}
    public  void addCust(String Name){}

}
```

To illustrate the use of an interface in this example, an interface called `Nameable` is defined:

Click here to view code image

```
public interface Nameable {

    public abstract String getName();
    public abstract void setName(String name);

}
```

We could potentially have a large number of different implementations, but all the rest of the code (the application) is the same. In this small example, the code savings might not look like a lot. But in a large, real-world application, the code savings is significant. Let's take a look at the donut shop implementation:

Click here to view code image

```
public class DonutShop extends Shop implements Nameable {

    String companyName;

    String[] menuItems = {
        "Donuts",
        "Muffins",
        "Danish",
        "Coffee",
        "Tea"
    };

    public String[] getInventory() {

        return menuItems;

    }

    public void buyInventory(String item) {
```

```
        System.out.println("\nYou have just purchased " +
item);

    }

    public String getName(){

        return companyName;
    }

    public void setName(String name){

        companyName = name;
    }
}
```

The pizza shop implementation looks very similar:

Click here to view code image

```
public class PizzaShop extends Shop implements Nameable {

    String companyName;

    String[] foodOfferings = {
        "Pizza",
        "Spaghetti",
        "Garden Salad",
        "Antipasto",
        "Calzone"
    }

    public String[] getInventory() {

        return foodOfferings;

    }

    public void buyInventory(String item) {

        System.out.println("\nYou have just purchased " +
item);

    }

    public String getName(){
```

```
        return companyName;
    }

    public void setName(String name){

        companyName = name;
    }

}
```

Unlike the initial case, where a large number of customized applications exist, we now have only a single primary class (`Shop`) and various customized classes (`PizzaShop`, `DonutShop`). There is no coupling between the application and any of the customized classes. The only thing the application is coupled to is the contract (`Shop`). The contract specifies that any implementation of `Shop` must provide an implementation for two methods, `getInventory()` and `buyInventory()`. It also must provide an implementation for `getName()` and `setName()` that relates to the interface `Nameable` that is implemented.

Although this solution solves the problem of highly coupled implementations, we still have the problem of deciding which implementation to use. With the current strategy, we would still have to have separate applications. In essence, you must provide one application for each `Shop` implementation. Even though we are using the `Shop` contract, we still have the same situation as before we used the contract:

```
DonutShop myShop= new DonutShop();

PizzaShop myShop = new PizzaShop ();
```

How do we get around this problem? We can create objects dynamically. In Java, we can use code like this:

Click here to view code image

```
String className = args[0];

Shop  myShop;

myShop = (Shop)Class.forName(className).newInstance();
```

In this case, you set `className` by passing a parameter to the code. (There are other ways to set `className`, such as by using a system property.)

Let's look at `Shop` using this approach. (Note that there is no exception handling and nothing else besides object instantiation.)

Click here to view code image

```
class TestShop {

   public static void main (String args[]) {

      Shop shop = null;

      String className = args[0];

      System.out.println("Instantiate the class:" + className
+ "\n");

      try {

       // new pizzaShop();
         shop = (Shop)Class.forName(className).newInstance();

      } catch (Exception e) {

         e.printStackTrace();
      }

      String[] inventory = shop.getInventory();

      // list the inventory

      for (int i=0; i<inventory.length; i++) {
         System.out.println("Argument" + i + " = " +
inventory[i]);
      }

      // buy an item

      shop.buyInventory(Inventory[1]);

   }

}
```

In this way, we can use the same application code for both `PizzaShop` and `DonutShop`. If we add a `GroceryShop` application, we only have to provide the implementation and the appropriate string to the main application. No application code needs to change.

Conclusion

When designing classes and object models, it is vitally important to understand how the objects are related to each other. This chapter discusses the primary topics of building objects: inheritance, interfaces, and composition. In this chapter, you have learned how to build reusable code by designing with contracts.

In Chapter 9, "Building Objects and Object-Oriented Design," we complete our OO journey and explore how objects that might be totally unrelated can interact with each other.

References

Booch, Grady and Robert A. Maksimchuk and Michael W. Engel and Bobbi J. Young and Jim Conallen and Kelli A. Houston. 2007. *Object-Oriented Analysis and Design with Applications*, Third Edition. Boston, MA: Addison-Wesley.

Coad, Peter, and Mark Mayfield. 1997. *Java Design*. Upper Saddle River, NJ: Prentice Hall.

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

9. Building Objects and Object-Oriented Design

The previous two chapters cover the topics of inheritance and composition. In Chapter 7, "Mastering Inheritance and Composition," we learned that inheritance and composition represent the primary ways to build objects. In Chapter 8, "Frameworks and Reuse: Designing with Interfaces and Abstract Classes," we learned that there are varying degrees of inheritance and how inheritance, interfaces, abstract classes, and composition all fit together.

This chapter covers the issue of how objects are related to each other in an overall design. You might say that this topic was already introduced, and you would be correct. Both inheritance and composition represent ways in which objects interact. However, inheritance and composition have one significant difference in the way objects are built. When inheritance is used, the end result is, at least conceptually, a single class that incorporates all the behaviors and attributes of the inheritance hierarchy. When composition is used, one or more classes are used to build another class.

Although it is true that inheritance is a relationship between two classes, what is really happening is that a parent is created that incorporates the attributes and methods of a child class. Let's revisit the example of the `Person` and `Employee` classes (see Figure 9.1).

Although there are indeed two separately designed classes here, the relationship is not simply interaction—it is inheritance. Basically, an employee is a person. An `Employee`object does not send a message to a `Person` object. An `Employee` object does need the services of a `Person` object. This is because an `Employee` object is a `Person` object.

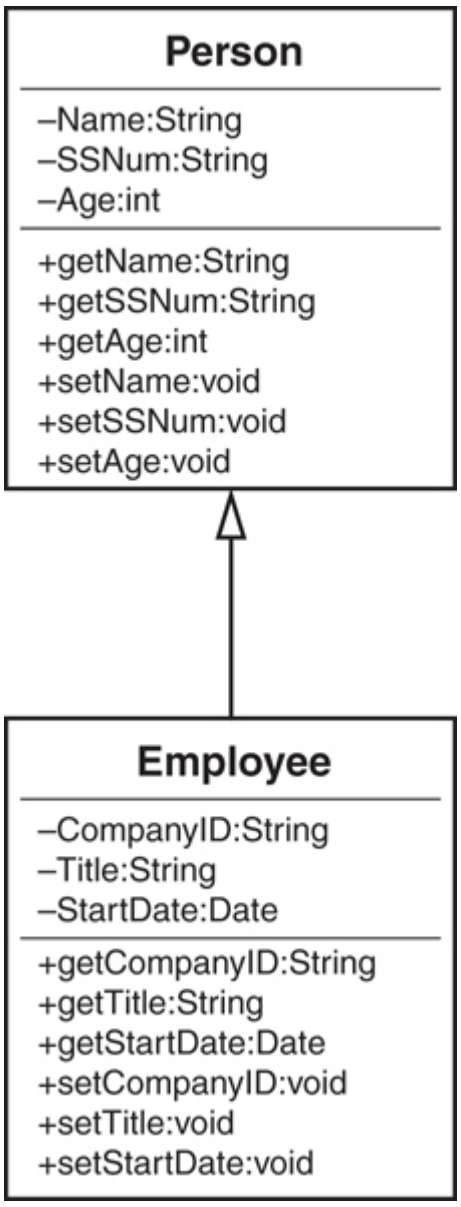

Figure 9.1 An inheritance relationship.

However, composition is a different situation. Composition represents interactions between distinct objects. So, whereas Chapter 8 primarily covers the different flavors of inheritance, this chapter delves into the various flavors of composition and how objects interact with each other.

Composition Relationships

We have already seen that composition represents a part of a whole. Although the inheritance relationship is stated in terms of is-a, composition is

stated in terms of has-a. We know intuitively that a car "has-a" steering wheel (see Figure 9.2).

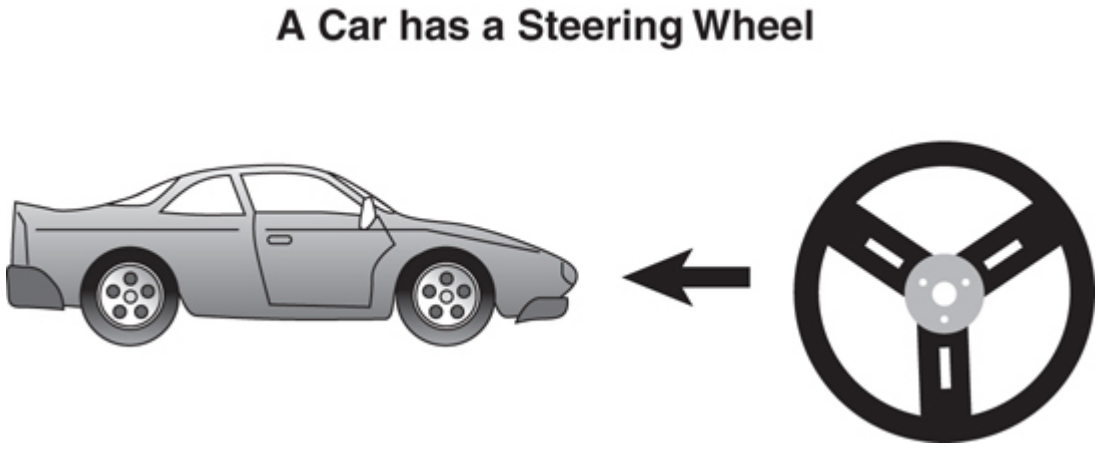

Figure 9.2 A composition relationship.

> ### Is-a and Has-a
>
> Please forgive my grammar: For consistency, I will stick with "has a engine," even though "has an engine" is grammatically correct. I do this because I want to simply state the rules as "is-a" and "has-a."

The reason to use composition is that it builds systems by combining less complex parts. This is a common way for people to approach problems. Studies show that even the best of us can keep, at most, seven chunks of data in our short-term memory at one time. Thus, we like to use abstract concepts. Instead of saying that we have a large unit with a steering wheel, four tires, an engine, and so on, we say that we have a car. This makes it easier for us to communicate and keep things clear in our heads.

Composition also helps in other ways, such as making parts interchangeable. If all steering wheels are the same, it does not matter which specific steering wheel is installed in a specific car. In software development, interchangeable parts mean reuse.

In Chapters 7 and 8 of their book *Object-Oriented Design in Java*, Stephen Gilbert and Bill McCarty present many examples of associations and composition in much more detail. I highly recommend referencing this material for a more in-depth look into these subjects. Here we address some

of the more fundamental points of these concepts and explore some variations of their examples.

Building in Phases

Another major advantage in using composition is that systems and subsystems can be built independently, and perhaps more importantly, tested and maintained independently.

There is no question that software systems are quite complex. To build quality software, you must follow one overriding rule to be successful: Keep things as simple as possible. For large software systems to work properly and be easily maintained, they must be broken into smaller, more manageable parts. How do you accomplish this? In a 1962 article titled "The Architecture of Complexity," Nobel Prize winner Herbert Simon noted the following thoughts regarding stable systems:

- **"Stable complex systems usually take the form of a hierarchy, where each system is built from simpler subsystems, and each subsystem is built from simpler subsystems still.**—You might already be familiar with this principle because it forms the basis for functional decomposition, the method behind procedural software development. In object-oriented design, you apply the same principles to composition—building complex objects from simpler pieces.
- **"Stable, complex systems are nearly decomposable."**—This means you can identify the parts that make up the system and can tell the difference between interactions between the parts and inside the parts. Stable systems have fewer links between their parts than they have inside their parts. Thus, a modular stereo system, with simple links between the speakers, turntable, and amplifier, is inherently more stable than an integrated system, which isn't easily decomposable.
- **"Stable complex systems are almost always composed of only a few different kinds of subsystems, arranged in different**

combinations."—Those subsystems, in turn, are generally composed of only a few different kinds of parts.

- **"Stable systems that work have almost always evolved from simple systems that worked."**—Rather than build a new system from scratch—reinventing the wheel—the new system builds on the proven designs that went before it.

In our stereo example (see Figure 9.3), suppose the stereo system was totally integrated and was not built from components (that is, the stereo system was one big black-box system). In this case, what would happen if the CD player broke and became unusable? You would have to take in the entire system for repair. Not only would this be more complicated and expensive, but you would not have the use of any of the other components.

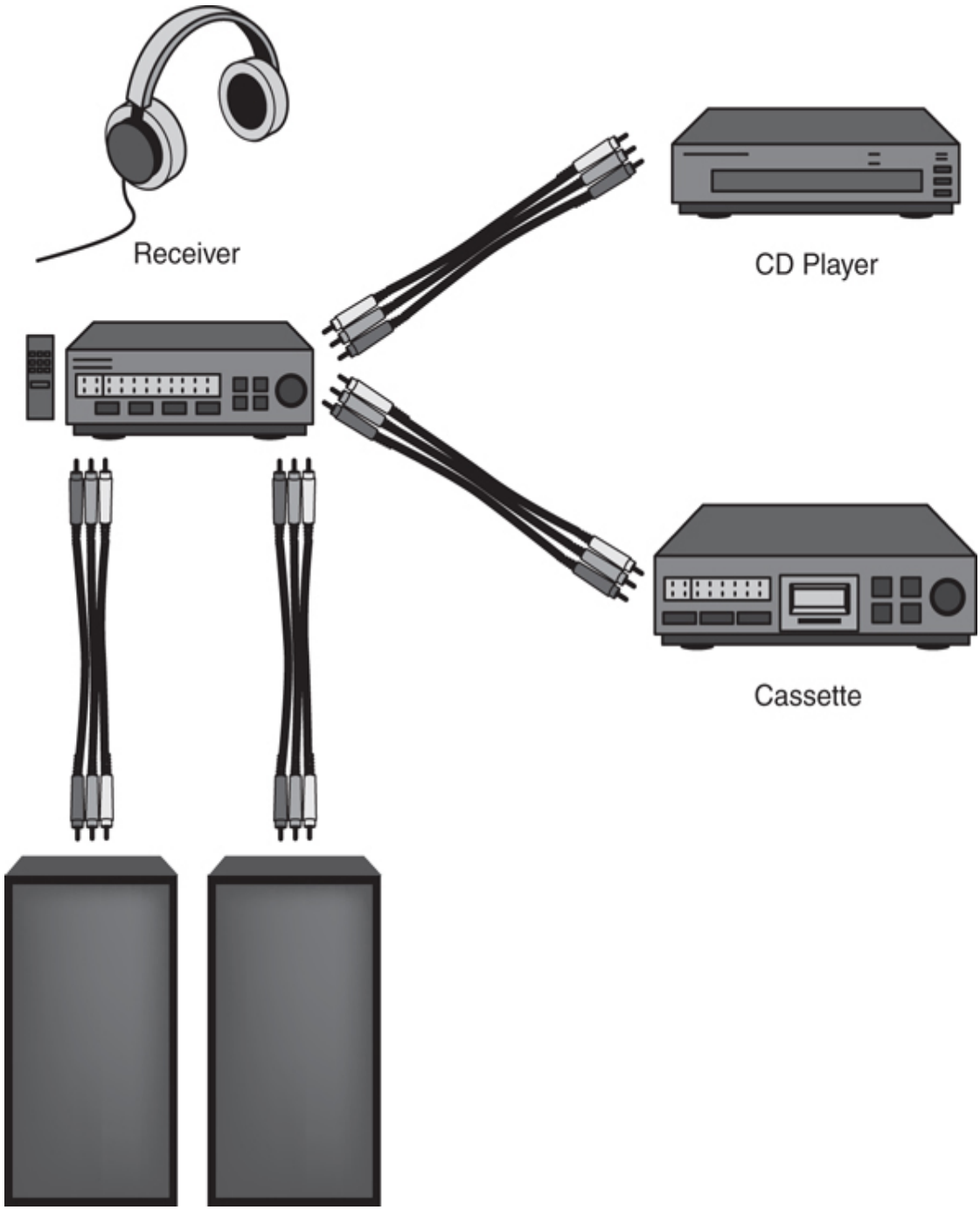

Figure 9.3 Building, testing, and verifying a complete system one step at a time.

This concept becomes very important to languages such as Java and those included in the .NET framework. Because objects are dynamically loaded, decoupling the design is quite important. For example, if you distribute a Java application and one of the class files needs to be re-created (for bug

fixes or maintenance), you would be required to redistribute only that particular class file. If all code was in a single file, the entire application would need to be redistributed.

Suppose the system is broken into components rather than a single unit. In this case, if the CD player broke, you could disconnect the CD player and take it in for repair. (Note that all the components are connected by patch cords.) This would be less complicated and less expensive, and it would take less time than having to deal with a single, integrated unit. As an added benefit, you could still use the rest of the system. You could even buy another CD player because it is a component. The repairperson could then plug your broken CD player into his repair systems to test and fix it. All in all, the component approach works quite well. Composition is one of the primary strategies that you, as a software designer, have in your arsenal to fight software complexity.

One major advantage of using components is that you can use components that were built by other developers within the organization, or even third-party vendors. However, using a software component from another source requires a certain amount of trust. Third-party components must come from a reliable source, and you must feel comfortable that the software is properly tested, not to mention that it must perform the advertised functions properly. There are still many who would rather build their own than trust components built by others.

Types of Composition

Generally, there are two types of composition: association and aggregation. In both cases, these relationships represent collaborations between the objects. The stereo example we just used to explain one of the primary advantages of composition represents an association.

> ### Is Composition a Form of Association?
>
> Composition is another area in OO technologies where there is a question of which came first, the chicken or the egg. Some texts say that composition is a form of association, and some say that an association is a form of composition. In any event, in this book, we

> consider inheri-tance and composition the two primary ways to build classes. Thus, in this book, association is considered a form of composition.

All forms of composition include a has-a relationship. However, subtle differences exist between associations and aggregations based on how you visualize the parts of the whole. In an aggregation, you normally see only the whole, and in associations, you normally see the parts that make up the whole.

Aggregations

Perhaps the most intuitive form of composition is aggregation. Aggregation means that a complex object is composed of other objects. A TV set is a clean, neat package that you use for entertainment. When you look at your HD TV, you see a single unit. Most of the time, you do not stop to think about the fact that the HD TV contains some microchips, a screen, a tuner, and so on. Sure, you see a switch to turn the set on and off, and you certainly see the picture screen. However, this is not the way people normally think of HD TVs. When you go into an appliance store, the salesperson does not say, “Let me show you this aggregation of microchips, a picture screen, a tuner, and so on.” The salesperson says, “Let me show you this HD TV.”

Similarly, when you go to buy a car, you do not pick and choose all the individual components of the car. You do not decide which spark plugs to buy or which door handles to buy. You go to buy a car. Of course, you do choose some options, but for the most part, you choose the car as a whole, a complex object made up of many other complex and simple objects (see Figure 9.4).

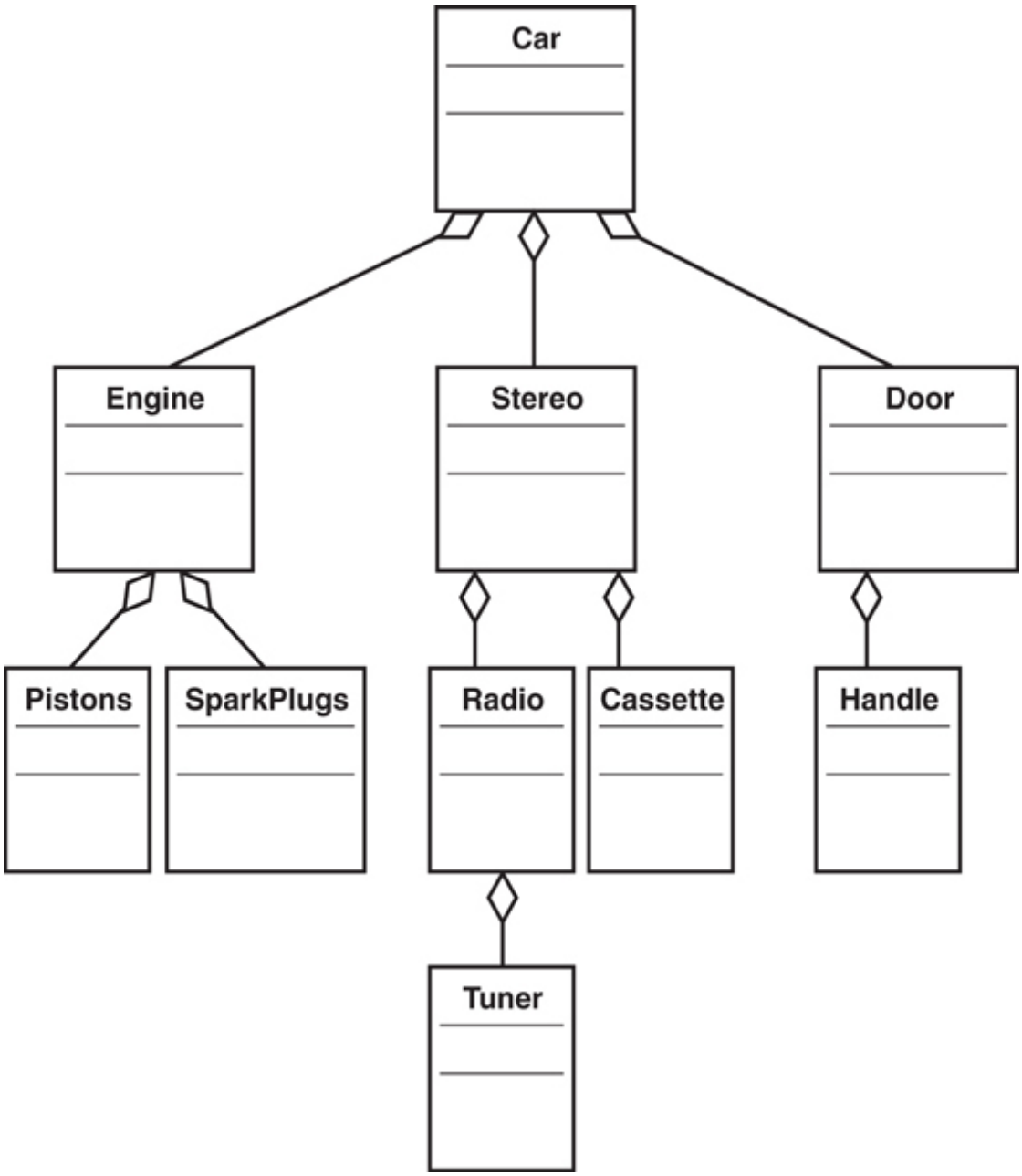

Figure 9.4 An aggregation hierarchy for a car.

Associations

Whereas aggregations represent relationships where you normally see only the whole, associations present both the whole and the parts. As stated in the stereo example, the various components are presented separately and connect to the whole by use of patch cords (the cords that connect the various components).

Consider a traditional desktop computer system as an example (see Figure 9.5); the whole is the computer system. The components are the monitor, keyboard, mouse, and main box. Each is a separate object, but together they represent the whole of the computer system. The main computer is using the keyboard, the mouse, and the monitor to delegate some of the work. In other words, the computer box needs the service of a mouse but does not have the capability to provide this service by itself. Thus, the computer box requests the service from a separate mouse via the specific port and cable connecting the mouse to the box.

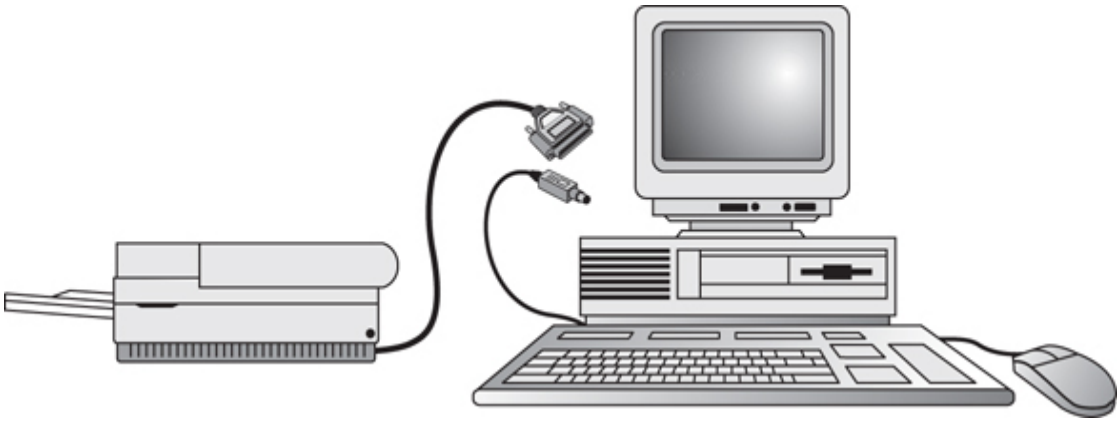

Figure 9.5 Associations as a separate service.

> ### Aggregation Versus Association
>
> An aggregation is a complex object composed of other objects. An association is used when one object wants another object to perform a service for it.

Using Associations and Aggregations Together

One thing you might have noticed in all the examples is that the dividing lines between what is an association and what is an aggregation are often blurred. Suffice it to say that many of your most interesting design decisions will come down to whether to use associations or aggregations.

For example, the desktop computer system example used to describe associations also contains some aggregation. Although the interaction between the computer box, the monitor, the keyboard, and the mouse is association, the computer box itself represents aggregation. You see only the

computer box, but it is actually a complex system made up of other objects, including chips, motherboards, video cards, and so on.

Consider that an `Employee` object might be composed of an `Address` object and a `Spouse` object. You might consider the `Address` object as an aggregation (basically a part of the `Employee` object), and the `Spouse` object as an association. To illustrate, suppose both the employee and the spouse are employees. If the employee is fired, the spouse is still in the system but the association is broken.

Similarly, in the stereo example, the receiver has an association with the speakers as well as the CD. Yet, the speakers and the CD are themselves aggregations of other objects, such as power cords.

In the car example, although the engine, spark plugs, and doors represent composition, the stereo also represents an association relationship. In reality, cars and desktop computers are a mix of aggregations *and* associations.

> **No One Right Answer**
>
> As usual, there isn't a single, absolutely correct answer when it comes to making a design decision. Design is not an exact science. Although we can make general rules to live by, these rules are not hard and fast.

Avoiding Dependencies

When using composition, it is desirable to avoid making objects highly dependent on one another. One way to make objects very dependent on each other is to mix domains. In the best of all worlds, an object in one domain should not be mixed with an object in another domain, except under certain circumstances. We can return again to the stereo example to explain this concept.

By keeping all the components in separate domains, the stereo system is easier to maintain. For example, if the CD component breaks, you can send the CD player off to be repaired individually. In this case, the CD player and the MP3 player have separate domains. This provides flexibility, such as

buying the CD player and the MP3 player from separate manufacturers. So, if you decide you want to swap out the CD player with a brand from another manufacturer, you can.

Sometimes there is a certain convenience in mixing domains. A good example of this, one that I have been using for years, pertains to the existence of the old-style TV/VCR combinations. Granted, it is convenient to have both in the same module. However, if the TV breaks, the VCR is unusable—at least as part of the unit it was purchased in.

You need to determine what is more important in specific situations: whether you want convenience or stability. There is no right answer. It all depends on the application and the environment. In the case of the TV/VCR combination, we decided that the convenience of the integrated unit far outweighed the risk of lower unit stability (see Figure 9.6). Revisit the stereo system in Figure 9.3 to reinforce what a non-integrated system looks like.

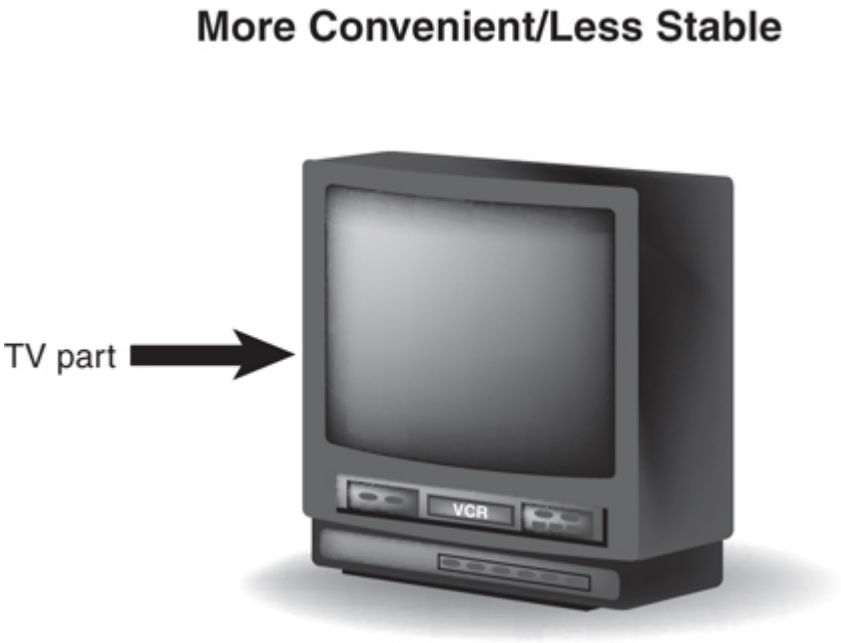

Figure 9.6 Convenience versus stability.

Interfaces solve this and managing dependencies is a major part of this. If interfaces are defined in a shared library and implementations are defined in more concrete classes, you can afford to mix domains by using the behavior contracts.

> **Mixing Domains**
>
> The convenience of mixing domains is a design decision. If the power of having a TV/VCR com-bination outweighs the risk and potential downtime of the individual components, the mixing of domains may well be the preferred design choice.

Cardinality

In their book *Object-Oriented Design in Java*, Gilbert and McCarty describe cardinality as the number of objects that participate in an association and whether the participation is optional or mandatory. To determine cardinality, Gilbert and McCarty ask the following questions:

- Which objects collaborate with which other objects?
- How many objects participate in each collaboration?
- Is the collaboration optional or mandatory?

For example, let's consider the following example. We are creating an `Employee` class that inherits from `Person` and has relationships with the following classes:

- `Division`
- `JobDescription`
- `Spouse`
- `Child`

What do these classes do? Are they optional? How many does an `Employee` need?

- `Division`
    - This object contains the information relating to the division that the employee works for.

    - Each employee must work for a division, so the relationship is mandatory.
    - The employee works for one, and only one, division.

- `JobDescription`
    - This object contains a job description, most likely containing information such as salary grade and salary range.
    - Each employee must have a job description, so the relationship is mandatory.
    - The employee can hold various jobs during the tenure at a company. Thus, an employee can have many job descriptions. These descriptions can be kept as a history if an employee changes jobs, or it is possible that an employee might hold two different jobs at one time. For example, a supervisor might take on an employee's responsibilities if the employee quits and a replacement has not yet been hired.

- `Spouse`
    - In this simplistic example, the `Spouse` class contains only the anniversary date.
    - An employee can be married or not married. Thus, a spouse is optional.
    - An employee can have only one spouse.

- `Child`
    - In this simple example, the `Child` class contains only the string `FavoriteToy`.
    - An employee can have children or not have children.
    - An employee can have no children or an infinite number of children (wow!). You could make a design decision as to the upper limit of the number of children that the system can handle.

To sum up, Table 9.1 represents the cardinality of the associations of the classes we just considered.

Table 9.1 Cardinality of Class Associations

Optional/Association	Cardinality	Mandatory
Employee/Division	1	Mandatory
Employee/JobDescription	1 . . n	Mandatory
Employee/Spouse	0 . . 1	Optional
Employee/Child	0 . . n	Optional

> ### Cardinality Notation
>
> The notation of 0 . . 1 means that an employee can have either zero or one spouse. The notation of 0 . . n means that an employee can have any number of children from zero to an unlimited number. The *n* basically represents infinity.

Figure 9.7 shows the class diagram for this system. Note that in this class diagram, the cardinality is indicated along the association lines. Refer to Table 9.1 to see whether the association is mandatory.

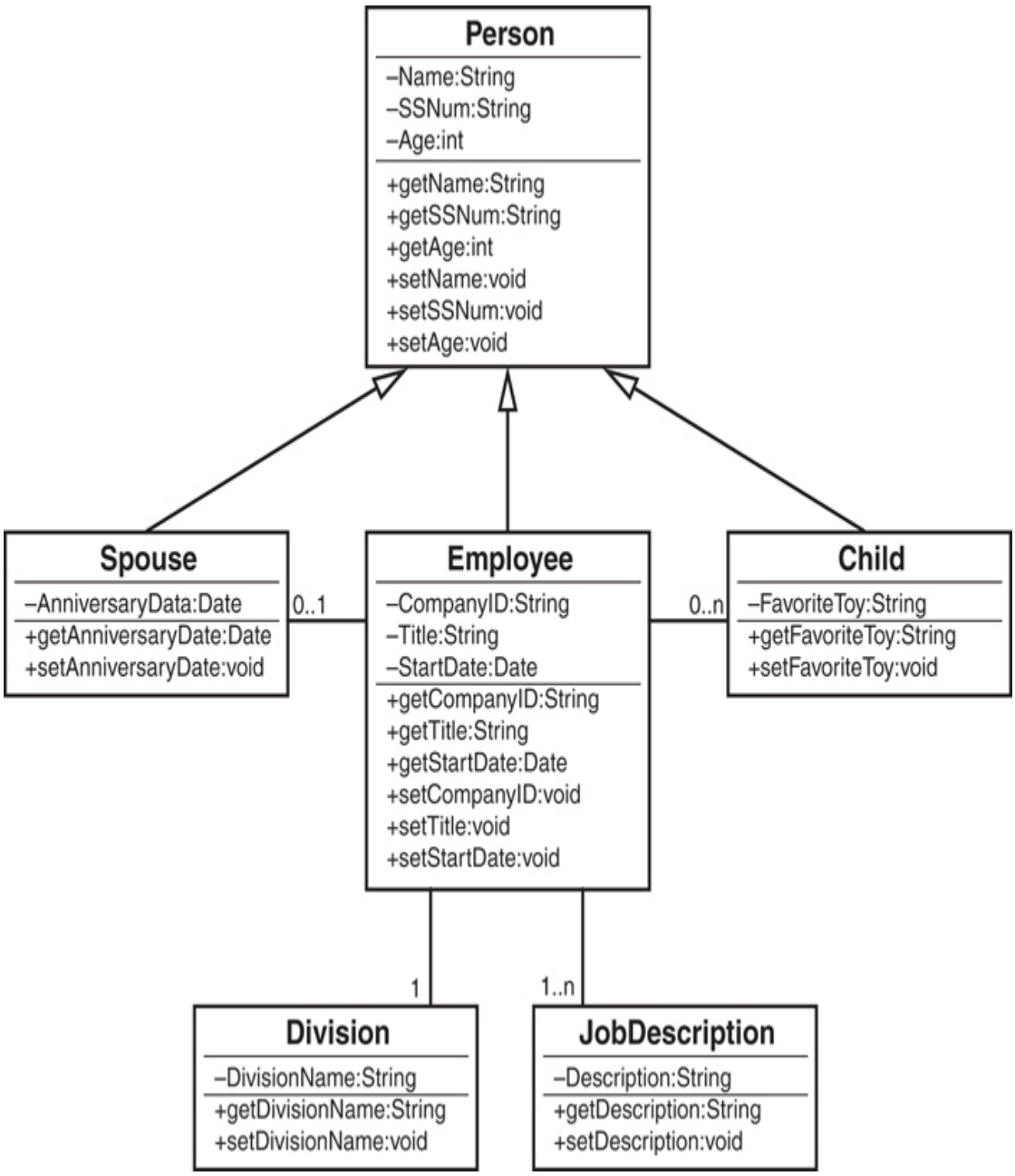

Figure 9.7 Cardinality in a UML diagram.

Multiple Object Associations

How do we represent an association that might contain multiple objects (such as 0 to many
children) in code? Here is the code for the `Employee` class:

Click here to view code image

```
import java.util.Date;

public class Employee extends Person{

        private String CompanyID;
        private String Title;
        private Date StartDate;

        private Spouse spouse;
        private Child[] child;
        private Division division;
        private JobDescription[] jobDescriptions;

        public String getCompanyID() {return CompanyID;}
        public String getTitle() {return Title;}
        public Date getStartDate() {return StartDate;}

        public void setCompanyID(String CompanyID) {}
        public void setTitle(String Title) {}
        public void setStartDate(int StartDate) {}

}
```

Note that the classes that have a one-to-many relationship are represented by arrays in the code:

```
private Child[] child;
private JobDescription[] jobDescriptions;
```

Optional Associations

One of the most important issues when dealing with associations is to make sure that your application is designed to check for optional associations. This means that your code must check to see whether the association is `null`.

Suppose in the previous example that your code assumes that every employee has a spouse. However, if one employee is not married, the code will have a problem (see Figure 9.8). If your code does indeed expect a spouse to exist, it may well fail and leave the system in an unstable state. The bottom line is that the code must check for a `null` condition, and must handle this as a valid condition.

Figure 9.8 Checking all optional associations.

For example, if no spouse exists, the code must not attempt to invoke a spouse behavior. This could lead to an application failure. Thus, the code must be able to process an `Employee` object that has no spouse.

Tying It All Together: An Example

Let's work on a simple example that will tie the concepts of inheritance, interfaces, composition, associations, and aggregations together into a single, short system diagram.

Consider the example used in Chapter 8, with one addition: We will add an `Owner` class that will take the dog out for walks.

Recall that the `Dog` class inherits directly from the `Mammal` class. The solid arrow represents this relationship between the `Dog` class and the `Mammal` class in Figure 9.9. The `Nameable` class is an interface that `Dog`

implements, which is represented by the dashed arrow from the `Dog` class to the `Nameable` interface.

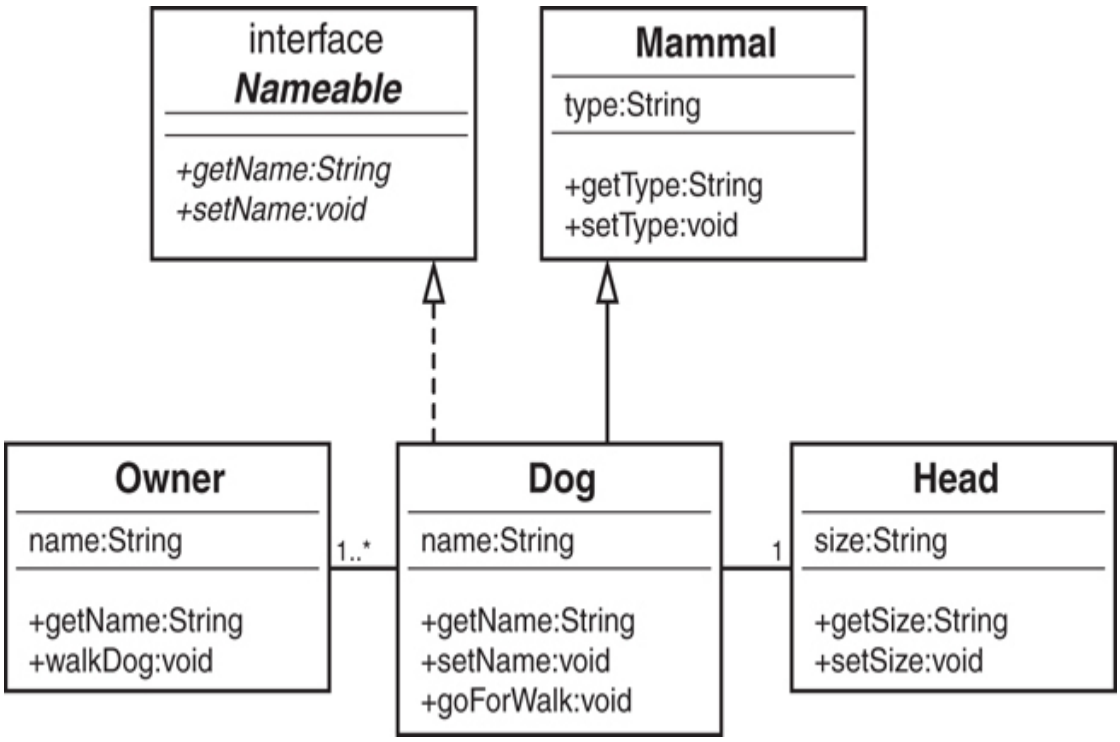

Figure 9.9 A UML diagram for the `Dog` example.

In this chapter, we are mostly concerned with associations and aggregations. The relationship between the `Dog` class and the `Head` class is considered aggregation because the head is actually part of the dog. The cardinality on the line connecting the two class diagrams specifies that a dog can have only a single head.

The relationship between the `Dog` class and the `Owner` class is association. The owner is clearly not part of the dog, or vice versa, so we can safely eliminate aggregation. However, the dog does require a service from the owner—the act of taking him on a walk. The cardinality on the line connecting the `Dog` and `Owner` classes specifies that a dog can have one or more owners (for example, a wife and husband can both be considered owners, with shared responsibility for walking the dog).

These relationships—inheritance, interfaces, composition, associations, and aggregations— represent the bulk of the design work you will encounter when designing OO systems.

> ### Where Is the Head?
>
> You might decide that it makes sense to attach the Head class to the `Mammal` class instead of the Dog class, because all mammals supposedly have a head. For this model, I was using the Dog class as the focal point of the example, so that is why I attached the Head to the Dog itself.

Conclusion

In this chapter, we have explored some of the finer points of composition and its two primary types: aggregation and association. Whereas inheritance represents a new kind of already existing object, composition represents the interactions between various objects.

The past three chapters have covered the basics of inheritance and composition. Using these concepts and your skills in the software development process, you are on your way to designing solid classes and object models. Chapter 10, "Design Patterns," explores how to use UML class diagrams to assist in the modeling of object models.

References

Booch, Grady and Robert A. Maksimchuk and Michael W. Engel and Bobbi J. Young and Jim Conallen and Kelli A. Houston. 2007. *Object-Oriented Analysis and Design with Applications*, Third Edition. Boston, MA: Addison-Wesley.

Coad, Peter, and Mark Mayfield. 1997. *Java Design*. Upper Saddle River, NJ: Prentice Hall.

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

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

10. Design Patterns

One of the interesting things about software development is that when you create a software system, you are actually modeling a real-world system. For example, in the Information Technology industry, it is safe to say that IT *is* the business—or at least IT *implements* the business. To write the business software systems, the developers must thoroughly understand the business models. As a result, the developers often have the most intimate knowledge of a company's business processes.

We have seen this concept throughout this book as it relates to our educational discussions. For example, when we discussed using inheritance to abstract out the behaviors and attributes of mammals, the model was based on the true real-life model, not a contrived model that we created for our own purposes.

Thus, when we create a mammal class, we can use it to build countless other classes, such as dogs and cats and so on, because all mammals share certain behaviors and attributes. This works when we study dogs, cats, squirrels, and other mammals because we can see patterns. These patterns allow us to inspect an animal and make the determination that it is indeed a mammal, or perhaps a reptile, which would have other patterns of behaviors and attributes.

Throughout history, humans have used patterns in many aspects of life, including engineering. These patterns go hand-in-hand with the holy grail of software development: software reuse. In this chapter, we consider design patterns, a relatively new area of software development (the seminal book on design patterns was published in 1995).

Design patterns are perhaps one of the most influential developments that have come out of the object-oriented movement in the past several years. Patterns lend themselves perfectly to the concept of reusable software

development. Because object-oriented development is all about reuse, patterns and object-oriented development go hand-in-hand.

The basic concept of design patterns revolves around the principle of best practices. By *best practices*, we mean that when good and efficient solutions are created, these solutions are documented in a way that others can benefit from previous successes—as well as learn from the failures.

One of the most important books on object-oriented software development is *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. This book was an important milestone for the software industry and has become so entrenched in the computer science lexicon that the book's authors have become known as the Gang of Four. In writings on object-oriented topics, you will often see the Gang of Four referred to as the GoF.

The intent of this chapter is to explain what design patterns are. (Explaining each design pattern is far beyond the scope of this book and would take more than one volume.) To accomplish this, we explore each of the three categories of design patterns (creational, structural, and behavioral) as defined by the Gang of Four and provide a concrete example of one pattern in each category.

Why Design Patterns?

The concept of design patterns did not necessarily start with the need for reusable software. In fact, the seminal work on design patterns is about constructing buildings and cities. As Christopher Alexander noted in *A Pattern Language: Towns, Buildings, Construction*, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use the solution a million times over, without ever doing it the same way twice."

> ### The Four Elements of a Pattern
>
> The GoF describe a pattern as having four essential elements:

- The *pattern name* is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary. It lets us design at a higher level of abstraction. Having a vocabulary for patterns lets us talk about them with our colleagues, in our documentation, and even to ourselves. It makes it easier to think about designs and to communicate them and their trade-off to others. Finding good names has been one of the hardest parts of developing our catalog.?
- The *problem* describes when to apply the pattern. It explains the problem and its content. It might describe specific design problems, such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.
- The *solution* describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many situations. Instead, the pattern provides an abstract description of a design problem, and how a general arrangement of elements (classes and objects in our case) solves it.
- The *consequences* are the results and trade-offs of applying the pattern. Although con-sequences are often unvoiced, when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of the applying pattern. The consequences for software often concern space and time trade-offs. They might address language and implementation issues as well. Because reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing the consequences explicitly helps you understand and evaluate them.

Smalltalk's Model/View/Controller

For historical perspective, we need to consider the Model/View/Controller (MVC) introduced in Smalltalk (and used in other object-oriented languages). MVC is often used to illustrate the origins of design patterns. The Model/View/Controller paradigm was used to create user interfaces in Smalltalk. Smalltalk was perhaps the first *popular* object-oriented language.

> ### Smalltalk
>
> Smalltalk is the result of several great ideas that emerged from Xerox PARC. These ideas included the mouse and using a windowing environment, among others. Smalltalk is a wonderful language that provided the foundation for all the object-oriented languages that followed. One of the complaints about C++ is that it's not really object-oriented, whereas Smalltalk is. Although C++ had a larger following in the early days of OO, Smalltalk has always had a very dedicated core group of supporters. Java is a mostly OO language that embraced the C++ developer base.

Design Patterns defines the MVC components in the following manner:

> The Model is the application object, the View is the screen presentation, and the Controller defines the way the user interface reacts to user input.

The problem with previous paradigms is that the Model, View, and Controller used to be lumped together in a single entity. For example, a single object would have included all three of the components. With the MVC paradigm, these three components have separate and distinct interfaces. So if you want to change the user interface of an application, you only have to change the View. Figure 10.1 illustrates what the MVC design looks like.

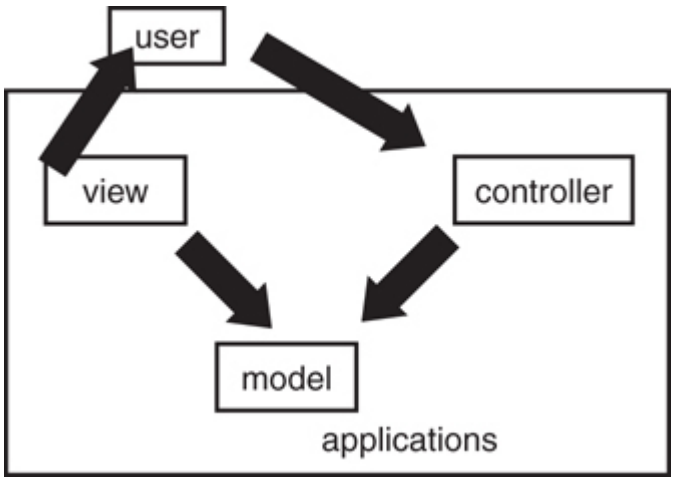

Figure 10.1 Model/View/Controller paradigm.

Remember that much of what we have been learning about object-oriented development has to do with interfaces versus implementation. As much as possible, we want to separate the interface from the implementation. We also want to separate interface from interface as much as possible. For example, we do not want to combine multiple interfaces that do not have anything to do with one another (or the solution to the problem at hand). The MVC was one of the early pioneers in this separation of interfaces. The MVC explicitly defines the interfaces between specific components pertaining to a very common and basic programming problem—the creation of user interfaces and their connection to the business logic and data behind them.

If you follow the MVC concept and separate the user interface, business logic, and data, your system will be much more flexible and robust. For example, assume that the user interface is on a client machine, the business logic is on an application server, and the data is located on a data server. Developing your application in this way would allow you to change the way the GUI looks without having an impact on the business logic or the data. Likewise, if your business logic changes and you calculate a specific field differently, you can change the business logic without having to change the GUI. And finally, if you want to swap databases and store your data differently, you can change the way the data is stored on the data server without affecting either the GUI or the business logic. This assumes, of course, that the interfaces between the three do not change.

MVC Example

> One example is that of a listbox used in a user interface. Consider a GUI that includes a list of phone numbers. The listbox is the view, the phone list is the model, and the controller is the logic that binds the listbox to the phone list.

> ### MVC Drawbacks
>
> Although the MVC is a great design, it can be somewhat complex in that a lot of attention must be paid to the upfront design. This is a problem with object-oriented design in general—there is a fine line between a good design and a cumbersome design. The question remains: How much complexity should you build into the system with regard to a complete design?

Types of Design Patterns

Design Patterns features 23 patterns grouped into the three categories that follow. Most of the examples are written in C++, with some written in Smalltalk. The time of the book's publication is indicative of the use of C++ and Smalltalk. The publication date of 1995 was right at the cusp of the Internet revolution and the corresponding popularity of the Java programming language. After the benefit of design patterns became apparent, many other books rushed in to fill the newly created market.

In any event, the actual language used is irrelevant. *Design Patterns* is inherently a design book, and the patterns can be implemented in any number of languages. The authors of the book divided the patterns into three categories:

- *Creational patterns* create objects for you, rather than having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case.
- *Structural patterns* help you compose groups of objects into larger structures, such as complex user interfaces or accounting data.
- *Behavioral patterns* help you define the communication between objects in your system and how the flow is controlled in a complex program.

The following sections discuss one example from each of these categories to provide a flavor of design patterns. For a comprehensive list and description of individual design patterns, refer to the books listed at the end of this chapter.

Creational Patterns

The creational patterns consist of the following categories:

- Abstract factory
- Builder
- Factory method
- Prototype
- Singleton

As stated earlier, the scope of this chapter is to describe what a design pattern is—not to describe each and every pattern in the GoF book. Thus, we will cover a single pattern in each category. With this in mind, let's consider an example of a creational pattern and look at the factory pattern.

The Factory Method Design Pattern

Creating, or instantiating, objects may well be one of the most fundamental concepts in object oriented programming. It goes without saying that you can't use an object unless that object exists. When writing code, the most obvious way to instantiate an object is to use the `new` keyword.

To illustrate, let's revisit the `Shape` example used throughout this book. Here we have the familiar parent class `Shape`, which is abstract, and the child class `Circle`, which is the concrete implementation. We instantiate a `Circle` class in the usual way by employing the `new` keyword:

```
abstract class Shape {

}

class Circle extends Shape {

}
```

```
Circle circle = new Circle();
```

Although this code certainly works, there may be many other places in your code where you need to instantiate a `Circle`, or any other `Shape` for that matter. In many cases, you will have specific object creation parameters that need to be handled each time you create a `Shape`.

As a result, any time you change the way objects are created, the code must be changed in every location where a `Shape` object is instantiated. The code is highly coupled because a change in one location necessitates code changes in potentially many other locations. Another problem with this approach is that it exposes the object creation logic to the programmers using the classes.

To remedy these situations, we can implement a factory method. In short, the factory method is responsible for encapsulating all instantiation so that it is uniform across the implementation. You use the factory to instantiate, and the factory is responsible for instantiating properly.

Factory Method Pattern

The fundamental intent of the factory method pattern is to create objects without having to specify the exact class—in effect, using interfaces to create new types of objects.

To illustrate how to implement a factory pattern, let's create a factory for the `Shape` class example. The class diagram in Figure 10.2 helps visualize how the various classes in the example interact.

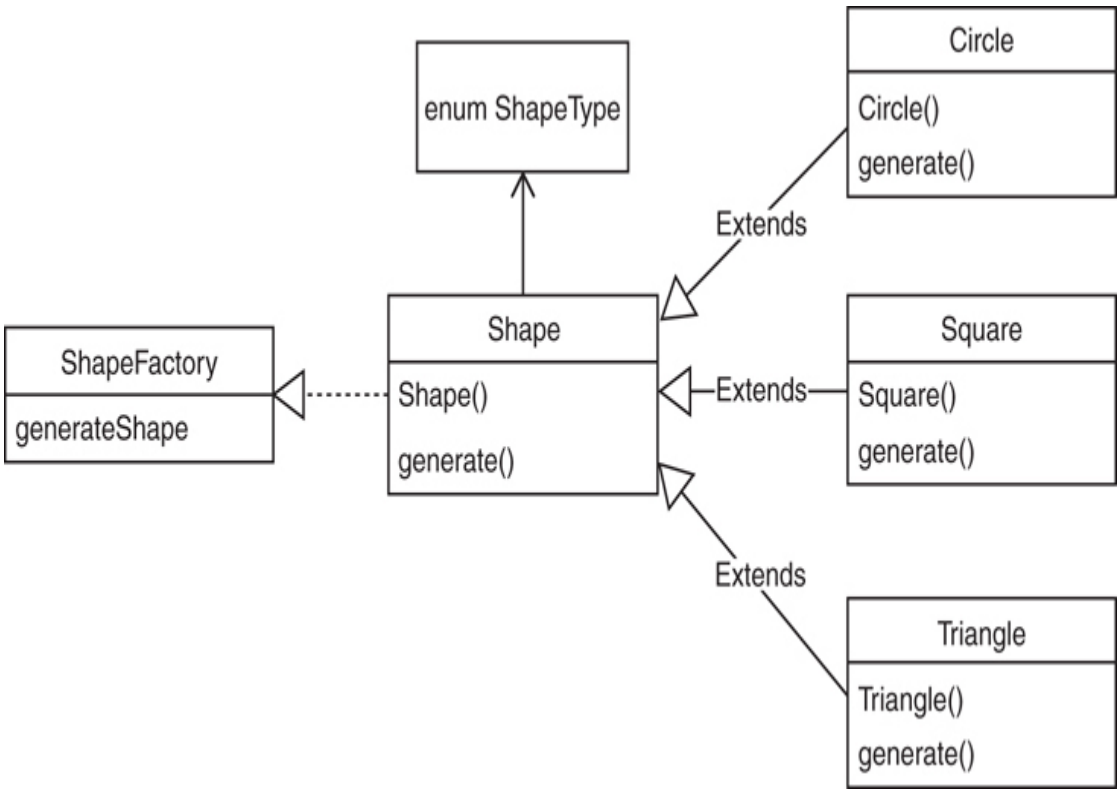

Figure 10.2 Creating a factory for the `Shape` class.

In some ways, you can think of a factory as a wrapper. Consider the fact that there may be some significant logic involved in instantiating an object and you don't want the programmer (user) to be concerned with this logic. It is almost like the concept of an accessor method (getters and setters) when the retrieval of a value is inside some logic (like when a password is required). Using a factory method is useful when you don't know ahead of time which specific class you might need. For example, you may know that a shape is required, but you don't know the specific shape (at least not yet). With this in mind, all possible classes must be in the same hierarchy; that is, all the classes in this example must be a subclass of `Shape`. In fact, a factory is used precisely because you don't know what you need, allowing you to add some of the classes later. If you knew what you needed, you could simply "inject" the instance via a constructor or a setter method.

Basically, this is the definition of polymorphism.

We create an `enum` to contain the types of shapes. In this case, we will define `CIRCLE`, `SQUARE`, and `TRIANGLE`.

```
enum ShapeType {
    CIRCLE, SQUARE, TRIANGLE
}
```

We define the `Shape` class as abstract with just a constructor and an abstract method called `generate()`.

```
abstract class Shape {

    private ShapeType sType = null;

    public Shape(ShapeType sType) {
        this.sType = sType;
    }

    // Generate the shape
    protected abstract void generate();

}
```

The child classes, `CIRCLE`, `SQUARE`, and `TRIANGLE`, extend the `Shape` class, identify themselves, and provide the concrete implementation of the `generate()` method.

Click here to view code image

```
class Circle extends Shape {

    Circle() {
        super(ShapeType.CIRCLE);
        generate();
    }

    @Override
    protected void generate() {
        System.out.println("Generating a Circle");
    }
}

class Square extends Shape {

    Square() {
        super(ShapeType.SQUARE);
        generate();
    }
```

```
    @Override
    protected void generate() {
        System.out.println("Generating a Square");
    }
}

class Triangle extends Shape {

    Triangle() {
        super(ShapeType.TRIANGLE);
        generate();
    }

    @Override
    protected void generate() {
        System.out.println("Generating a Triangle");
    }
}
```

The `ShapeFactory` class, as the name implies, is the actual factory. Focus on the `generate()` method. While a `Factory` provides many advantages, note that the `generate()` method is the only location within the application that actually instantiates a `Shape`.

Click here to view code image

```
class ShapeFactory {
    public static Shape generateShape(ShapeType sType) {
        Shape shape = null;
        switch (sType) {

        case CIRCLE:
            shape = new Circle();
            break;

        case SQUARE:
            shape = new Square();
            break;

        case TRIANGLE:
            shape = new Triangle();
            break;

        default:
            // throw an exception
            break;
```

```
        }
        return shape;
    }
}
```

The traditional approach to instantiating these individual objects is to have the programmer directly instantiate the objects using the `new` keyword as follows:

Click here to view code image

```
public class TestFactoryPattern {
    public static void main(String[] args) {

        Circle circle = new Circle();
        Square square = new Square();
        Triangle triangle = new Triangle();

    }
}
```

However, properly using the factory requires that the programmer use the `ShapeFactory` class to obtain any `Shape` object:

Click here to view code image

```
public class TestFactoryPattern {
    public static void main(String[] args) {

        ShapeFactory.generateShape(ShapeType.CIRCLE);
        ShapeFactory.generateShape(ShapeType.SQUARE);
        ShapeFactory.generateShape(ShapeType.TRIANGLE);

    }
}
```

Structural Patterns

Structural patterns are used to create larger structures from groups of objects. The following seven design patterns are members of the structural category:

- Adapter

- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

As an example from the structural category, let's take a look at the adapter pattern. The adapter pattern is also one of the most important design patterns. This pattern is a good example of how the implementation and interface are separated.

The Adapter Design Pattern

The adapter pattern is a way for you to create a different interface for a class that already exists. The adapter pattern basically provides a class wrapper. In other words, you create a new class that incorporates (wraps) the functionality of an existing class with a new and—ideally—better interface. A simple example of a wrapper is the Java class `Integer`. The `Integer` class wraps a single `Integer` value inside it. You might wonder why you would bother to do this. Remember that in an object-oriented system, everything is an object. In Java, primitives, such as ints, floats, and so on, are not objects. When you need to perform functions on these primitives, such as conversions, you need to treat them as objects. Thus, you create a wrapper object and "wrap" the primitive inside it. Thus, you can take a primitive like the following:

```
int myInt = 10;
```

and wrap it in an `Integer` object:

```
Integer myIntWrapper = new Integer (myInt);
```

Now you can do a conversion, so you can treat it as a string:

```
String myString = myIntWrapper.toString();
```

This wrapper enables you to treat the original integer as an object, thus providing all the advantages of an object.

As for the adapter pattern itself, consider the example of a mail tool interface. Let's assume you have purchased some code that provides all the functionality you need to implement a mail client. This tool provides everything you want in a mail client, except you would like to change the interface slightly. In fact, all you want to do is change the API to retrieve your mail.

The following class provides a very simple example of a mail client for this example:

Click here to view code image

```
package MailTool;
public class MailTool {
   public MailTool () {
   }
   public int retrieveMail() {

    System.out.println ("You've Got Mail");

         return 0;
   }
}
```

When you invoke the `retrieveMail()` method, your mail is presented with the very original greeting "You've Got Mail." Now let's suppose you want to change the interface in all your company's clients from `retrieveMail()` to `getMail()`. You can create an interface to enforce this:

```
package MailTool;
interface MailInterface {
   int getMail();
}
```

You can now create your own mail tool that wraps the original tool and provide your own interface:

Click here to view code image

```
package MailTool;
class MyMailTool implements MailInterface {
   private MailTool yourMailTool;
   public MyMailTool () {
    yourMailTool= new MailTool();
        setYourMailTool(yourMailTool);
   }
   public int getMail() {
      return getYourMailTool().retrieveMail();
   }
   public MailTool getYourMailTool() {
      return yourMailTool ;
   }
   public void setYourMailTool(MailTool newYourMailTool) {
      yourMailTool = newYourMailTool;
   }
}
```

Inside this class, you create an instance of the original mail tool that you want to retrofit. This class implements `MailInterface`, which will force you to implement a `getMail()` method. Inside this method, you literally invoke the `retrieveMail()` method of the original mail tool.

To use your new class, you instantiate your new mail tool and invoke the `getMail()` method.

Click here to view code image

```
package MailTool;
public class Adapter
{
  public static void main(String[] args)
  {
    MyMailTool myMailTool = new MyMailTool();

    myMailTool.getMail();

  }
}
```

When you invoke the `getMail()` method, you are using this new interface to invoke the `retrieveMail()` method from the original tool. This is a very simple example; however, by creating this wrapper, you can enhance the interface and add your own functionality to the original class.

The concept of an adapter is quite simple, but you can create new and powerful interfaces using this pattern.

Behavioral Patterns

The behavioral patterns consist of the following categories:

- Chain of response
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method
- Visitor

As an example from the behavioral category, let's take a look at the iterator pattern. This is one of the most commonly used patterns and is implemented by several programming languages.

The Iterator Design Pattern

Iterators provide a standard mechanism for traversing a collection, such as a vector. Functionality must be provided so that each item of the collection can be accessed one at a time. The iterator pattern provides information hiding, keeping the internal structure of the collection secure. The iterator pattern also stipulates that more than one iterator can be created without interfering with each other. Java provides its own implementation of an iterator. The following code creates a vector and then inserts a number of strings into it:

Click here to view code image

```
package Iterator;

import java.util.*;
public class Iterator {
    public static void main(String args[]) {

        // Instantiate an ArrayList.
        ArrayList<String> names = new ArrayList();

        // Add values to the ArrayList
        names.add(new String("Joe"));
        names.add(new String("Mary"));
        names.add(new String("Bob"));
        names.add(new String("Sue"));

        //Now Iterate through the names
        System.out.println("Names:");
        iterate(names);
    }

    private static void iterate(ArrayList<String> arl) {
        for(String listItem : arl) {
                    System.out.println(listItem.toString());
                }
    }
}
```

Then we create an enumeration so that we can iterate through it. The method `iterate()` is provided to perform the iteration functionality. In this method, we use the Java enumeration method `hasMoreElements()`, which traverses the vector and lists all the names.

Antipatterns

Although a design pattern evolves from experiences in a positive manner, *antipatterns* can be thought of as collections of experiences that have gone awry. It is well documented that most software projects are ultimately deemed unsuccessful. In fact, as indicated in the article “Creating Chaos” by Johnny Johnson, fully one-third of all projects are cancelled outright. It would seem obvious that many of these failures are caused by poor design decisions.

The term *antipattern* derives from the fact that design patterns are created to proactively solve a specific type of problem. An antipattern, on the other hand, is a reaction to a problem and is gleaned from bad experiences. In short, whereas design patterns are based on solid design practices, antipatterns can be thought of as practices to avoid.

In the November 1995 *C++ Report*, Andrew Koenig described two facets of antipatterns:

- Those that describe a bad solution to a problem, which result in a bad situation.
- Those that describe how to get out of a bad situation and how to proceed from there to a good solution.

Many people believe that antipatterns are more useful than design patterns. This is because antipatterns are designed to solve problems that have already occurred. This boils down to the concept of root-cause analysis. A study can be conducted with data that might indicate why the original design, perhaps an actual design pattern, did not succeed. It might be said that antipatterns emerge from the failure of previous solutions. Thus, antipatterns have the benefit of hindsight.

For example, in his article "Reuse Patterns and Antipatterns," Scott Ambler identifies a pattern called a *robust artifact* and defines it as follows:

> An item that is well-documented, built to meet general needs instead of project-specific needs, thoroughly tested, and has several examples to show how to work with it. Items with these qualities are much more likely to be reused than items without them. A Robust Artifact is an item that is easy to understand and work with.

However, there are certainly many situations when a solution is declared reusable and then no one ever reuses it. Thus, to illustrate an antipattern, he writes:

> Someone other than the original developer must review a Reuseless Artifact to determine whether or not anyone might be interested in it. If so, the artifact must be reworked to become a Robust Artifact.

Thus, antipatterns lead to the revision of existing designs, and the continuous refactoring of those designs until a workable solution is found.

> **Some Good Examples of Antipatterns**
>
> - Singleton
> - Service locator
> - Magic strings/magic numbers
> - Interface bloat
> - Coding by exception
> - Error hiding/swallowing

Conclusion

In this chapter, we explored the concept of design patterns. Patterns are part of everyday life, and this is just the way you should be thinking about object-oriented designs. As with many things pertaining to information technology, the roots for solutions are founded in real-life situations.

Although this chapter covered design patterns only briefly, you should explore this topic in greater detail by picking up one of the books referenced at the end of this chapter.

References

Alexander, Christopher, et al. 1977. *A Pattern Language: Towns, Buildings, Construction*. Cambridge, UK: Oxford University Press.

Ambler, Scott. “Reuse Patterns and Antipatterns.” *2000 Software Development Magazine.*

Gamma, Erich, et al. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley.

Grand, Mark. 2002. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML*, Second Edition, volume 1. Hoboken, NJ: Wiley.

Jaworski, Jamie. 1999. *Java 2 Platform Unleashed*. Indianapolis, IN: Sams Publishing.

Johnson, Johnny. "Creating Chaos." *American Programmer*, July 1995.

Larman, Craig. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development,* Third Edition. Hoboken, NJ: Wiley.

11. Avoiding Dependencies and Highly Coupled Classes

As presented in Chapter 1, "Introduction to Object-Oriented Concepts," the traditional criteria of classical object-oriented programming are encapsulation, inheritance, and polymorphism. Theoretically, to consider a programming language as an object-oriented language, it must follow these three principles. In addition, as also covered in Chapter 1, I like to include composition.

Thus, when I teach object-oriented programming, my list of fundamental concepts looks like this:

- Encapsulation
- Inheritance
- Polymorphism
- Composition

> **Tip**
>
> Perhaps I should add interfaces to this list, but I have always considered interfaces to be a specific type of inheritance.

Adding composition to this list is even more important in today's development environment because of the debate over how to use inheritance appropriately. Concerns about using inheritance are not a recent phenomenon. In the past several years, this debate has heated up. Many developers I talk to advocate for using composition rather than inheritance (often called composition over inheritance). In fact, some avoid using inheritance at all, or at least limit the use of inheritance to a single hierarchical level.

The reason for focusing on how to use inheritance revolves around the issue of coupling. The arguments for using inheritance are, most certainly, reusability, extensibility, and polymorphism; however, inheritance can cause problems by creating dependencies between classes—in effect, coupling the classes. These dependencies create potential problems for maintenance and testing. Chapter 7, "Mastering Inheritance and Composition," discussed how inheritance might actually weaken encapsulation, which seems counterintuitive because they are both fundamental concepts. Nevertheless, this is actually part of the fun, and requires that we really think about how we should use inheritance.

> **Caution**
>
> Be aware that I am not advocating avoiding inheritance. The discussion here is actually about avoiding dependencies and highly coupled classes. When to use inheritance is an important part of this discussion.

This debate leads to the following question: if not inheritance, then what? The short answer is to use composition. This should not be surprising because throughout the book I contend that there are really only two ways to reuse classes: using inheritance and using composition. You can either create a child from a parent class via inheritance or contain one class within another class using composition.

If, as some people advocate, inheritance is to be avoided, why do we spend time learning it? The answer is simple: A lot of code utilizes inheritance. As most developers soon come to understand, the vast majority of the code encountered appears in maintenance mode. Thus, it is imperative to understand how to fix, enhance, and maintain code written using inheritance. You may even write some new code using inheritance. In short, a programmer needs to cover all the possible bases and learn the entire developers' toolkit. However, this also means that we have to keep adding tools to that kit as well as rethink how we use them.

Again, please understand that I am not making any value judgments here. I am not claiming that inheritance is problematic and to avoid it. What I am saying is that it is important to fully grasp how inheritance is used, carefully study

alternative ways of design, and then decide for yourself. Thus, the intent of the examples in this chapter is not necessarily to describe the optimal way to design your classes; they are educational exercises meant to get you thinking about the issues associated with deciding between inheritance and composition. Remember that it is important for all technologies to evolve, keep the good, and refine the not-so good.

Moreover, composition poses its own coupling issues. In Chapter 7 I discussed the various types of composition: associations and aggregations. Aggregations are objects that are embedded in other objects (created with the `new` keyword) while associations are objects that are passed into other objects via a parameter list. Because aggregations are embedded in objects, they are highly coupled, which we want to avoid.

Therefore, while inheritance may have obtained a reputation as encouraging highly coupled classes, composition (using aggregations) also can create highly coupled classes. Let's revisit the stereo component example used in Chapter 9, "Building Objects and Object-Oriented Design," to bring all of these concepts together in a specific example.

Creating a stereo with aggregations can be likened to creating a boombox, which is a product that has all the components embedded inside a single unit. In many situations, this can be very convenient. It can be picked up, moved easily, and requires no special assembly. However, this design can also lead to many problems. If one component, say the MP3 player, breaks, you must take in the entire unit for repair. Even worse, many problems may arise to render the entire boombox unusable, such as an electrical issue.

Creating a stereo with associations can mitigate many of the problems encountered with aggregations. Think of a component stereo system as a bunch of associations connected by patch cords (or wireless). In this design, there is a central object called a receiver connected to several other objects such as speakers, CD players, even turntables and cassette players. In fact, think of this as a vendor-neutral solution because we can simply obtain a component off the shelf, which is a major advantage.

In this situation, if the CD player breaks, you simply disconnect it, providing the opportunity to either fix the CD player (while still enjoying the use of the other components) or swapping it out with a new CD player that works. This

is the advantage of using associations and keeping the coupling between classes to a minimum.

> **Tip**
>
> As pointed out in Chapter 9, although highly coupled classes are generally frowned upon, there might be times when you are willing to accept the risk of a highly coupled design. The boombox is one such example. Despite the fact that it has a highly coupled design, it is sometimes the preferred choice.

Now that we have reviewed the coupling issues of both inheritance and composition, let's explore examples of some highly coupled designs using both inheritance and composition. As I often do in the classroom, we will iterate through these examples until we use a technique called dependency injection to mitigate the coupling issues.

Composition versus Inheritance and Dependency Injection

To begin, we can focus on how to take an inheritance model (gleaned from examples often used in this book) and redesign it, not with inheritance but with composition. The second example shows how we can redesign with composition—albeit using aggregation, which is not necessarily an optimal solution. The third example shows how to avoid aggregations and design with associations instead—the concept of *dependency injection*.

1) Inheritance

Whether or not you buy into the argument of composition over inheritance, let's begin by presenting a straightforward example of inheritance and explore how it might otherwise be implemented using composition, revisiting the mammal example used throughout the book.
In this case, we introduce a bat—a mammal that can fly, as seen in Figure 11.1.

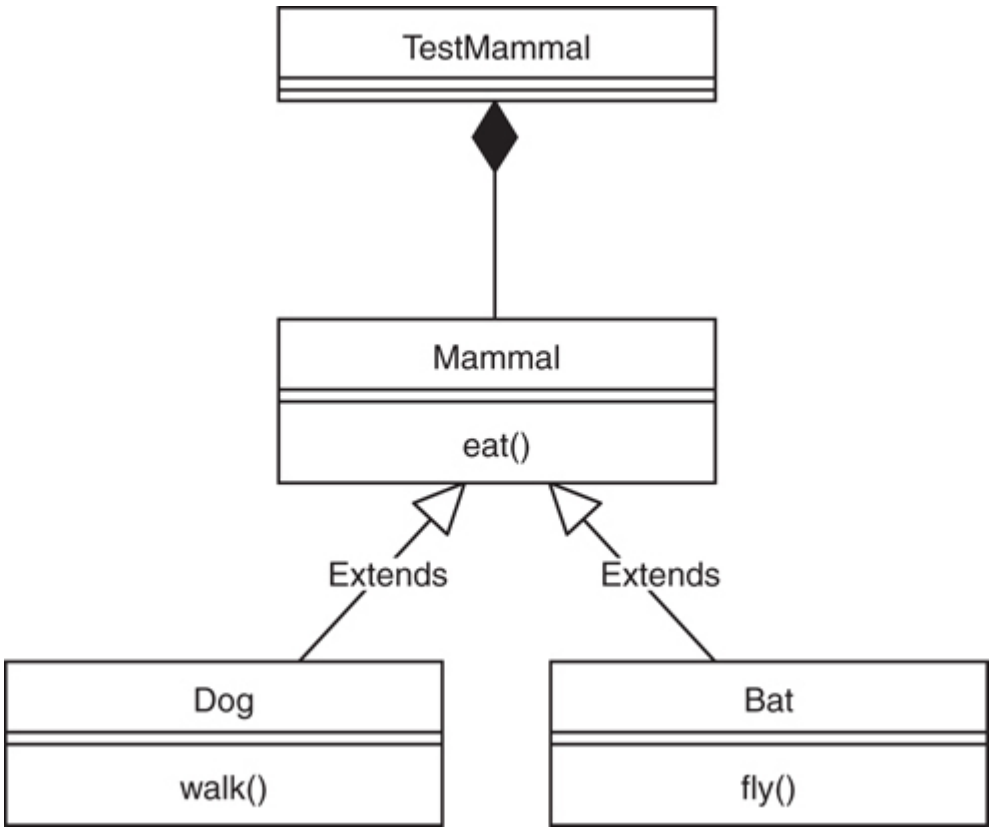

Figure 11.1 Using inheritance to create mammals.

In this example specifically, inheritance appears to be the obvious choice. Creating a `Dog` class that inherits from `Mammal` is a slam dunk—isn't it? Look at the following code, which utilizes inheritance in this manner:

Click here to view code image

```
class Mammal {
    public void eat () {System.out.println("I am Eating");};
}
class Bat extends Mammal {
    public void fly () {System.out.println("I am Flying");};
}
class Dog extends Mammal {
    public void walk () {System.out.println("I am
Walking");};
}
public class TestMammal {

    public static void main(String args[]) {

        System.out.println("Composition over Inheritance");;
```

```
        System.out.println("\nDog");
        Dog fido = new Dog();
        fido.eat();
        fido.walk();
        System.out.println("\nBat");
        Bat brown = new Bat();
        brown.eat();
        brown.fly();
    }
}
```

In this design, a `Mammal` has a single behavior, `eat()`, assuming that all mammals must eat. However, we start to see the problem with inheritance immediately when we add two `Mammal` subclasses, `Bat` and `Dog`. While a dog can walk, not all mammals walk. In addition, while a bat can indeed fly, not all mammals fly. So the question is, where do these methods go? Just like in our earlier penguin example, because not all birds fly, deciding where to place methods in an inheritance hierarchy can be tricky.

Separating the `Mammal` class into `FlyingMammals` and `WalkingMammals` is not a very elegant solution because this is only the tip of the proverbial iceberg. Some mammals can swim, some mammals even lay eggs. Moreover, there are likely countless other behaviors that individual mammal species possess, and it might be impractical to create a separate class for all of these behaviors. Thus, rather than approaching this design as an *is-a* relationship, perhaps we should explore it using a *has-a* relationship.

2) Composition

In this strategy, rather than embedding the behaviors in the classes themselves, we create individual classes for each behavior. Therefore, rather than placing behaviors in an inheritance hierarchy, we can create classes for each behavior and then build individual mammals by including just the behaviors that they require (via aggregation).

Thus, we create a class called `Walkable` and a class called `Flyable`, as seen in Figure 11.2.

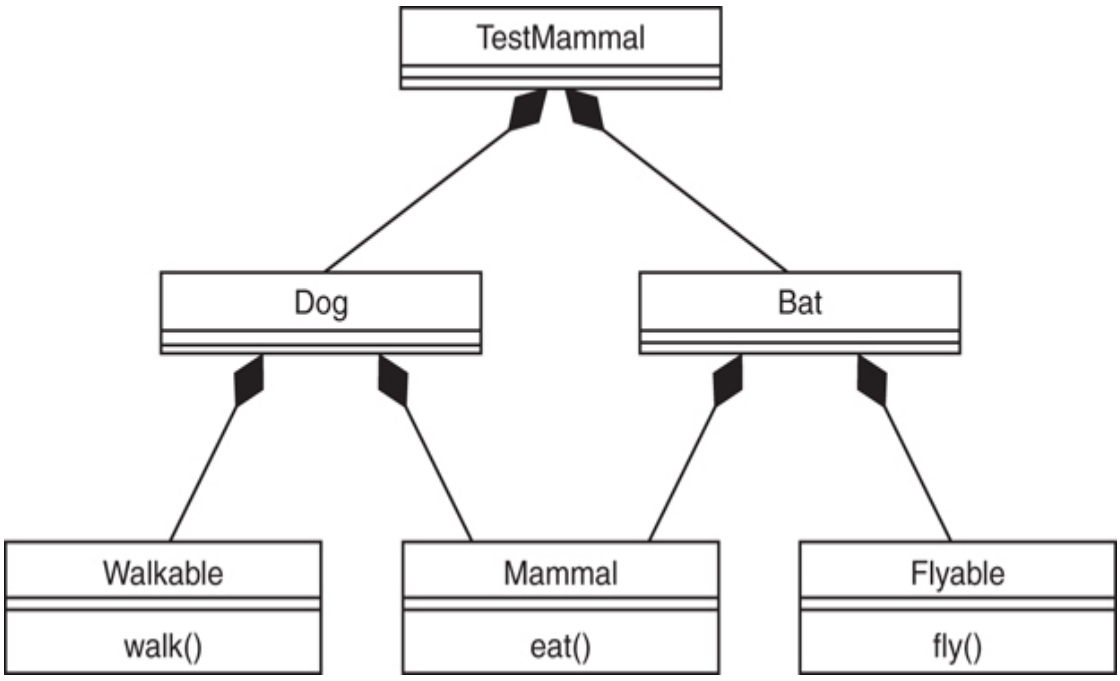

Figure 11.2 Using composition to create mammals.

For example, look at the following code. We still have the `Mammal` class with its `eat()` method, and we still have the `Dog` and `Bat` classes. The major design difference here is that the `Dog` and `Bat` classes obtain their behaviors via aggregation using composition.

> **Caution**
>
> Be aware that the term aggregation is used in the preceding paragraph. This example illustrates how composition can be used in lieu of inheritance; however, in this example, we are using aggregation, which still contains significant coupling. Thus, consider this an intermediate, educational step moving toward the next example using interfaces.

Click here to view code image

```
class Mammal {
    public void eat () {System.out.println("I am Eating");};
}
class Walkable {
    public void walk () {System.out.println("I am
Walking");};
}
```

```
class Flyable {
    public void fly () {System.out.println("I am Flying");};
}
class Dog {
    Mammal dog = new Mammal();
    Walkable walker = new Walkable();
}
class Bat {
    Mammal bat = new Mammal();
    Flyable flyer = new Flyable();
}
public class TestMammal {

    public static void main(String args[]) {

        System.out.println("Composition over Inheritance");;
        System.out.println("\nDog");;
        Dog fido = new Dog();
        fido.dog.eat();
        fido.walker.walk();

        System.out.println("\nBat");;
        Bat brown = new Bat();
        brown.bat.eat();
        brown.flyer.fly();

    }

}
```

> **Note**
>
> The intent of this example is to illustrate how to use composition in lieu of inheritance; that does not mean that you cannot use inheritance at all in your designs. If you determine that absolutely all mammals eat, then, for example, perhaps you would decide to place the eat() method in the Mammal class and have Dog and Bat inherit from Mammal. As always, this is a design decision.

Perhaps the heart of this discussion lies in the concept we covered earlier, that inheritance breaks encapsulation. This is easy to understand because a change in the `Mammal` class would require a recompilation (and perhaps even a redeployment) of all the `Mammal` subclasses. This means that the

classes are highly coupled, and this is counter to our stated goal of uncoupling classes as much as possible.

In our composition example, if we wanted to add a `Whale` class, none of the previously written classes would need a rewrite. You would add a class called `Swimmable` and a class called `Whale`. Then the `Swimmable` class could be reused for, say, a `Dolphin` class.

Click here to view code image

```
class Swimmable {
    public void fly () {System.out.println("I am
Swimming");};
}
class Whale {
    Mammal whale = new Mammal();
    Walkable swimmer = new Swimmable ();
}
```

The main application can add this functionality with no changes to the classes that previously existed.

```
System.out.println("\nWhale");
Whale shamu = new Whale();
shamu.whale.eat();
shamu.swimmer.swim();
```

One rule of thumb is to use inheritance in only truly polymorphic situations. Thus, `Circles` and `Rectangles` inheriting from `Shape` may well be a legitimate use of inheritance. On the other hand, behaviors such as walking and flying might not be good candidates for inheritance because overriding them could be problematic. For example, if you overrode the `fly()` method in `Dog`, the only obvious option would be a no-op (do nothing). Again, as we have seen with the earlier `Penguin` example, you don't want a `Dog` to run over a cliff, execute the available `fly()` method and then, to Fido's great chagrin, find that the `fly()` method doesn't do anything.

While this example does indeed implement this solution using composition, there is a serious flaw to the design. The objects are highly coupled, since the use of the `new` keyword is obvious.

```
class Whale {
    Mammal whale = new Mammal();
    Walkable swimmer = new Swimmable ();
}
```

To complete our exercise of decoupling the classes, we introduce the concept of *dependency injection*. In short, rather than creating objects inside other objects, we will inject the objects from the outside via parameter lists. The discussion focuses solely on the concept of injecting dependencies.

Dependency Injection

The example in the previous section uses composition (with aggregation) to provide the `Dog` with a behavior called `Walkable`. The `Dog` class literally created a new `Walkable` object within the `Dog` class itself, as the following code fragment illustrates:

```
class Dog {
    Walkable walker = new Walkable();
}
```

Although this does in fact work, the classes remain highly coupled. To completely decouple the classes in the previous example, let's implement the concept of dependency injection mentioned previously. Dependency injection and inversion of control are often covered together. One definition of inversion of control (IOC) is to make it someone else's responsibility to make an instance of the dependency and pass it to you. This is exactly what we will implement in this example.

Because not all mammals walk, fly, or swim, to begin the decoupling process, we create interfaces to represent the behaviors for our various mammals. For this example, I will focus on the walking behavior by creating an interface called `IWalkable` as seen in Figure 11.3.

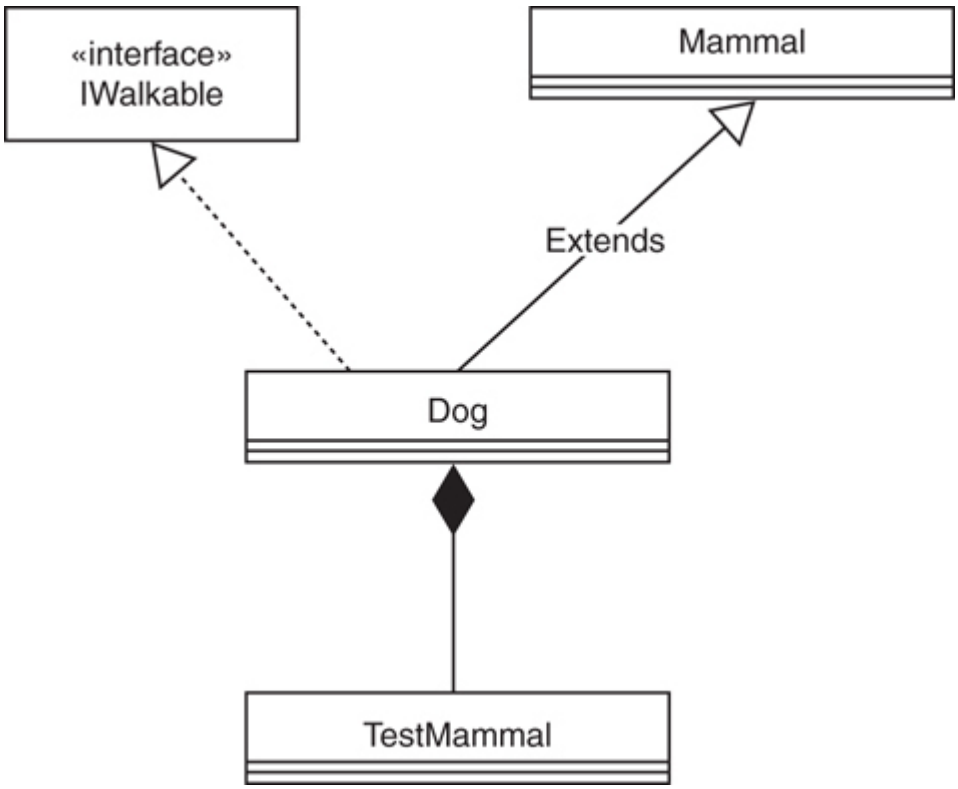

Figure 11.3 Using interfaces to create mammals.

The code for the `IWalkable` interface is as follows:

```
interface IWalkable {
    public void walk();
}
```

The only method in this interface is `walk()`, which is left to the concrete class to provide the implementation.

Click here to view code image

```
class Dog extends Mammal implements IWalkable{
    Walkable walker;
    public void setWalker (Walkable w) {
        this.walker=w;
    }
    public void walk () {System.out.println("I am
Walking");};
}
```

Note that the `Dog` class extends the `Mammal` class and implements the `IWalkable` interface. Also note that the `Dog` class provides a reference

and a constructor that provides the mechanism to inject the dependency.

```
Walkable walker;
public void setWalker (Walkable w) {
    this.walker=w;
}
```

In a nutshell, this is what dependency injection is. The `Walkable` behavior is not created inside the `Dog` class using the `new` keyword; it is injected into the `Dog` class via the parameter list.

Here is the complete example:

Click here to view code image

```
class Mammal {
    public void eat () {System.out.println("I am Eating");};
}
interface IWalkable {
    public void walk();
}
class Dog extends Mammal implements IWalkable{
    Walkable walker;
    public void setWalker (Walkable w) {
        this.walker=w;
    }
    public void walk () {System.out.println("I am
Walking");};
}
public class TestMammal {
    public static void main(String args[]) {
        System.out.println("Composition over Inheritance");
        System.out.println("\nDog");
        Walkable walker = new Walkable();
        Dog fido = new Dog();
        fido.setWalker(walker);
        fido.eat();
        fido.walker.walk();
    }
}
```

While this example uses injection by constructor, it is not the only way to handle dependency injection.

Injection by Constructor

One way to *inject* the `Walkable` behavior is to create a constructor within the `Dog` class that, when invoked, will accept an argument from the main application as follows:

```
class Dog {
    Walkable walker;
    public Dog (Walkable w) {
    this.walker=w;
    }
}
```

In this approach, the application instantiates a `Walkable` object and *inserts* it into the `Dog` via the constructor.

```
Walkable walker = new Walkable();

Dog fido = new Dog(walker);
```

Injection by Setter

Although a constructor will *initialize* attributes when an object is instantiated, there is often a need to reset values during the lifetime of an object. This is where accessor methods come into play—in the form of setters. The `Walkable` behavior can be *inserted* into the `Dog` class by using a setter, here called `setWalker()`:

```
class Dog {
    Walkable walker;
    public void setWalker (Walkable w) {
        this.walker=w;
    }
}
```

As with the constructor technique, the application instantiates a `Walkable` object and inserts it into the `Dog` via the setter:

```
Walkable walker = new Walkable();
Dog fido = new Dog();
fido.setWalker(walker);
```

Conclusion

Dependency injection decouples your class's construction from the construction of its dependencies. It is like buying something off the shelf (*from a vendor*) rather than building it on your own each time.

This plays to the heart of the discussion of Inheritance and composition. It is very important to note that this is simply a discussion. The purpose of this chapter is not necessarily to describe the "optimal" way to design your classes but to get you thinking about the issues associated with deciding between Inheritance and composition. In the next chapter, we explore The SOLID principles of object-oriented design, concepts highly regarded and accepted by the software development community.

References

Martin, Robert, et al. *Agile Software Development, Principles, Patterns, and Practices*. 2002. Boston: Pearson Education, Inc.

Martin, Robert, et al. *Clean Code*. 2009. Boston: Pearson Education, Inc.

12. The SOLID Principles of Object-Oriented Design

One of the most common statements that many developers make regarding object-oriented programming is that a primary advantage of OOP is that it models the real world. I admit that I use these words a lot when I discuss classical object-oriented concepts. According to Robert Martin (in at least one lecture that I viewed on YouTube), the idea that OO is closer to the way we think is simply marketing. Instead, he states that OO is about managing dependencies by inverting key dependencies to prevent rigid code, fragile code, and non-reusable code.

For example, in classical object-oriented programming courses, the practice often models the code directly to real-life situations. For example, if a dog *is-a* mammal, then this relationship is an obvious choice for inheritance. The strict *has-a* and *is-a* litmus test has been part of the OO mindset for years.

However, as we have seen throughout this book, trying to force an inheritance relationship can cause design problems (remember the barkless dog?). Is trying to separate barkless dogs from barking dogs, or flying birds from flightless birds, a smart inheritance design choice? Was this all put in place by object-oriented marketers? OK; forget the hype. As we saw in the previous chapter, perhaps focusing on a strict *has-a* and *is-a* decision is not necessarily the best approach. Perhaps we should focus more on decoupling the classes.

In the lecture I mentioned previously, Robert Martin, often referred to as Uncle Bob, defines these three terms to describe non-reusable code:

- **Rigidity**—When a change to one part of a program can break another part
- **Fragility**—When things break in unrelated places

- **Immobility**—When code cannot be reused outside its original context

SOLID was introduced to address these problems and strive to attain these goals. It defines five design principles that Robert Martin introduced to "make software designs more understandable, flexible, and maintainable." According to Robert Martin, though they apply to any object-oriented design, the SOLID principles can also form a core philosophy for methodologies such as agile development or adaptive software development. The SOLID acronym was introduced by
Michael Feathers.

The five SOLID principles are

- **SRP**—Single Responsibility Principle
- **OCP**—Open/Close Principle
- **LSP**—Liskov Substitution Principle
- **IPS**—Interface Segregation Principle
- **DIP**—Dependency Inversion Principle

This chapter focuses on covering these five principles and relates them to the classical object-oriented principles that have been in place for decades. My goal in covering SOLID is to explain the concepts in very simple examples. There is a lot of content online, including several very good YouTube videos. Many of these videos target developers, not necessarily students new to programming.

As I have attempted to do with all the examples in this book, my intent is not to get overly complicated but to distill the examples to the lowest common denominator for educational purposes.

The SOLID Principles of Object-Oriented Design

In Chapter 11, "Avoiding Dependencies and Highly Coupled Classes," we discussed some of the fundamental concepts leading up to our discussion of the five SOLID principles. In this chapter, we dive right in and cover each of

the SOLID principles in more detail. All SOLID definitions are from the Uncle Bob site: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

1) SRP: Single Responsibility Principle

The Single Responsibility Principle states that a class should have only a single reason to change. Each class and module in a program should focus on a single task. Thus, don't put methods that change for different reasons in the same class. If the description of the class includes the word "and," you might be breaking the SRP. In other words, every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated in the class.

Creating a shape hierarchy is one of the classic illustrations of inheritance. It is used often as a teaching example, and I use it a lot throughout this chapter (as well as the book). In this example, a `Circle` class inherits from an abstract `Shape` class. The `Shape` class provides an abstract method called `calcArea()` as the contract for the subclass. Any class that inherits from `Shape` must provide its own implementation of `calcArea()`:

```
abstract class Shape{
    protected String name;
    protected double area;
    public abstract double calcArea();
}
```

In this example, we have a `Circle` class that inherits from `Shape` and, as required, provides its implementation of `calcArea()`:

```
class Circle extends Shape{
    private double radius;

    public Circle(double r) {
        radius = r;
    }
    public double calcArea() {
        area = 3.14*(radius*radius);
        return (area);
    };
}
```

> **Caution**
>
> In this example, we are only going to include a Circle class to focus on the Single Responsibility Principle and keep the example as simple as possible.

A third class called `CalculateAreas` sums the areas of different shapes contained in a `Shape` array. The `Shape` array is of unlimited size and can contain different shapes, such as squares and triangles.

Click here to view code image

```
class CalculateAreas {
    Shape[] shapes;
    double sumTotal=0;
    public CalculateAreas(Shape[] sh){
        this.shapes = sh;
    }
    public double sumAreas() {
        sumTotal=0;
        for (inti=0; i<shapes.length; i++) {
            sumTotal = sumTotal + shapes[i].calcArea();
        }
        return sumTotal;
    }
    public void output() {
        System.out.println("Total of all areas = " +
sumTotal);
    }
}
```

Note that the `CalculateAreas` class also handles the output for the application, which is problematic. The area calculation behavior and the output behavior are coupled—contained in the same class.

We can verify that this code works with the following test application called `TestShape`:

Click here to view code image

```
public class TestShape {
    public static void main(String args[]) {

        System.out.println("Hello World!");
```

```
        Circle circle = new Circle(1);

        Shape[] shapeArray = new Shape[1];
        shapeArray[0] = circle;

        CalculateAreas ca = new CalculateAreas(shapeArray);

        ca.sumAreas();
        ca.output();
    }
}
```

Now with the test application in place, we can focus on the issue of the Single Responsibility Principle. Again, the issue is with the `CalculateAreas` class and that this class contains behaviors for *summing* the various areas as well as the *output*.

The fundamental point (and problem) here is this: If you want to change the functionality of the `output()` method, it requires a change to the `CalculateAreas` class regardless of whether the method for summing the areas changes. For example, if at some point we want to present the output to the console in HTML rather than in simple text, we must recompile and redeploy the code that sums the area because the responsibilities are coupled.

According to the Single Responsibility Principle, the goal is that a change to one method would not affect the other method, thus preventing unnecessary recompilations. "A class should have one, and only one, reason to change—a single responsibility to change."

To address this, we can put the two methods in separate classes, one for the original console output and one for the newly included HTML output:

Click here to view code image

```
class CalculateAreas {
    Shape[] shapes;
    double sumTotal=0;

    public CalculateAreas(Shape[] sh){
        this.shapes = sh;
    }
```

```
    public double sumAreas() {
        sumTotal=0;

        for (inti=0; i<shapes.length; i++) {

            sumTotal = sumTotal + shapes[i].calcArea();

        }

        return sumTotal;
    }
}
class OutputAreas {
    double areas=0;
    public OutputAreas(double a){
        this.areas = a;
    }
    public void console() {
        System.out.println("Total of all areas = " + areas);
    }

    public void HTML() {
        System.out.println("<HTML>");
        System.out.println("Total of all areas = " + areas);
        System.out.println("</HTML>");
    }
}
```

Now, using the newly written class, we can add functionality for HTML output without impacting the code for the area summing:

Click here to view code image

```
public class TestShape {
    public static void main(String args[]) {

        System.out.println("Hello World!");

        Circle circle = new Circle(1);

        Shape[] shapeArray = new Shape[1];
        shapeArray[0] = circle;

        CalculateAreas ca = new CalculateAreas(shapeArray);

        CalculateAreas sum = new CalculateAreas(shapeArray);
```

```
        OutputAreasoAreas = new OutputAreas(sum.sumAreas());

        oAreas.console();     // output to console
        oAreas.HTML();        // output to HTML

    }
}
```

The main point here is that you can now send the output to various destinations depending on requirements. If you want to add another output possibility, such as JSON, you can add it to the `OutputAreas` class without having to change the `CalculateAreas` class. As a result, you can redistribute the `CalculateAreas` class independently without having to do anything to the other classes.

2) OCP: Open/Close Principle

The Open/Close Principle states that you should be able to extend a class's behavior, without modifying it.

Let's revisit the shape example yet again. In the following code, we have a class called `ShapeCalculator` that accepts a `Rectangle` object, calculates the area of that object, and then returns that value. It is a simple application but it works only for rectangles.

Click here to view code image

```
class Rectangle{
    protected double length;
    protected double width;

    public Rectangle(double l, double w) {
        length = l;
        width = w;
    };
}
class CalculateAreas {

    private double area;

    public double calcArea(Rectangle r) {

        area = r.length * r.width;
```

```
        return area;

    }
}
public class OpenClosed {
    public static void main(String args[]) {

        System.out.println("Hello World");

        Rectangle r = new Rectangle(1,2);

        CalculateAreas ca = new CalculateAreas ();

        System.out.println("Area = "+ ca.calcArea(r));

    }
}
```

The fact that this application works only for rectangles brings us to a constraint that illustrates the Open/Closed Principle: If we want to add a `Circle` to the `CalculateArea` class (change what it does), we must change the module itself. Obviously, this is at odds with the Open/Closed Principle, which stipulates that we should not have to change the module to change what it does.

To comply with the Open/Closed Principle, we can revisit our tried and true shape example, where an abstract class called `Shape` is created and then all shapes must inherit from the `Shape` class, which has an abstract method called `getArea()`.

At this point, we can add as many different classes as we want without having to change the `Shape` class itself (for example, a `Circle`). We can now say that the `Shape` class is *closed*.

The following code implements this solution for a rectangle and a circle, and allows for the creation of unlimited shapes:

Click here to view code image

```
abstract class Shape {
    public abstract double getArea();
}
class Rectangle extends Shape
{
```

```
    protected double length;
    protected double width;

    public Rectangle(double l, double w) {
        length = l;
        width = w;
    };
    public double getArea() {
        return length*width;
    }

}
class Circle extends Shape
{
    protected double radius;

    public Circle(double r) {
        radius = r;
    };
    public double getArea() {
        return radius*radius*3.14;
    }
}
class CalculateAreas {
    private double area;

    public double calcArea(Shape s) {
        area = s.getArea();
        return area;
    }
}
public class OpenClosed {
    public static void main(String args[]) {

        System.out.println("Hello World");

        CalculateAreas ca = new CalculateAreas();

        Rectangle r = new Rectangle(1,2);

        System.out.println("Area = " + ca.calcArea(r));

        Circle c = new Circle(3);

        System.out.println("Area = " + ca.calcArea(c));

    }
}
```

Note that in this implementation, the `CalculateAreas()` method does not have to change when you add a new `Shape`.

You can scale your code without having to worry about legacy code. At its core, the Open/Closed Principle states that you should extend your code via subclasses and the original class does not need to be changed. However, the word *extension* is problematic in several discussions relating to SOLID. As we will cover in detail, if we are to favor composition over inheritance, how does this affect the Open/Closed Principle?

When following one of the SOLID principles, code may also comply with one of the other SOLID principles. For example, when designing to follow the Open/Closed Principle, the code may also comply with the Single Responsibility Principle.

3) LSP: Liskov Substitution Principle

The Liskov Substitution Principle states that the design must provide the ability to replace any instance of a parent class with an instance of one of its child classes. If a parent class can do something, a child class must also be able to do it.

Let's examine some code that might look reasonable but violates the Liskov Substitution Principle. In the following code, we have the typical abstract class called `Shape`. `Rectangle` then inherits from `Shape` and overrides its abstract method `calcArea()`. `Square`, in turn, inherits from `Rectangle`.

Click here to view code image

```
abstract class Shape{
    protected double area;

    public abstract double calcArea();
}
class Rectangle extends Shape{
    private double length;
    private double width;

    public Rectangle(double l, double w){
        length = l;
```

```
        width = w;
    }
    public double calcArea() {
        area = length*width;
        return (area);
    };

}
class Square extends Rectangle{
    public Square(double s){
        super(s, s);
    }
}

public class LiskovSubstitution {
    public static void main(String args[]) {

        System.out.println("Hello World");

        Rectangle r = new Rectangle(1,2);

        System.out.println("Area = " + r.calcArea());

        Square s = new Square(2);

        System.out.println("Area = " + s.calcArea());

    }
}
```

So far so good: a rectangle *is-a* shape so everything looks fine. Because a square *is-a* rectangle we are still fine—or are we?

Now we enter into a somewhat philosophical discussion: Is a square really a rectangle? Many people would say yes. However, while the square may well be a specialized type of a rectangle, it does have different properties than a rectangle. A rectangle is a parallelogram (opposite sides are congruent), as is a square. Yet, a square is also a rhombus (all sides are congruent), whereas a rectangle is not. Therefore, there are some differences.

The geometry is not really the issue when it comes to OO design. The issue is how we build rectangles and squares. Here is the constructor for the `Rectangle` class:

```
public Rectangle(double l, double w){
    length = l;
    width = w;
}
```

The constructor obviously requires two parameters. However, the `Square` constructor requires just one, even though its parent class, `Rectangle`, is expecting two.

```
class Square extends Rectangle{
    public Square(double s){
    super(s, s);
   }
```

In actuality, the functionality to compute area is subtly different for the two classes. In fact, the `Square` is kind of faking the `Rectangle` out by passing it the same parameter twice. This may seem like an acceptable workaround, but it really is something that may confuse someone maintaining the code and could very well cause unintended maintenance headaches down the road. This is an inconsistency at minimum and, perhaps, a questionable design decision. When you see a constructor calling another constructor, it might be a good idea to pause and reconsider the design—it might not be a proper child class.

How do you address this specific dilemma? Simply put, a square is not a *substitute* for a rectangle and should not be a child class. Thus, they should be separate classes.

Click here to view code image

```
 abstract class Shape {
    protected double area;

    public abstract double calcArea();
}

class Rectangle extends Shape {

    private double length;
    private double width;

    public Rectangle(double l, double w) {
        length = l;
        width = w;
```

```
    }

    public double calcArea() {
        area = length*width;
        return (area);
    };
}

class Square extends Shape {
    private double side;

    public Square(double s){
        side = s;
    }
    public double calcArea() {
        area = side*side;
        return (area);
    };
}
public class LiskovSubstitution {
    public static void main(String args[]) {

        System.out.println("Hello World");

        Rectangle r = new Rectangle(1,2);

        System.out.println("Area = " + r.calcArea());

        Square s = new Square(2);

        System.out.println("Area = " + s.calcArea());

    }
}
```

4) IPS: Interface Segregation Principle

The Interface Segregation Principle states that it is better to have many small interfaces than a few larger ones.

In this example, we are creating a *single* interface that includes *multiple* behaviors for a `Mammal`, `eat()` and `makeNoise()`:

Click here to view code image

```
interface IMammal {
    public void eat();
    public void makeNoise();
}
class Dog implements IMammal {
    public void eat() {
        System.out.println("Dog is eating");
    }
    public void makeNoise() {
        System.out.println("Dog is making noise");
    }
}
public class MyClass {
    public static void main(String args[]) {

        System.out.println("Hello World");

        Dog fido = new Dog();
        fido.eat();
        fido.makeNoise();
    }
}
```

Rather than creating a single interface for `Mammal`, we can create separate interfaces for all the behaviors:

Click here to view code image

```
interface IEat {
    public void eat();
}
interface IMakeNoise {
    public void makeNoise();
}
class Dog implements IEat, IMakeNoise {
    public void eat() {
        System.out.println("Dog is eating");
    }
    public void makeNoise() {
        System.out.println("Dog is making noise");
    }
}
public class MyClass {
    public static void main(String args[]) {

        System.out.println("Hello World");
```

```
        Dog fido = new Dog();
        fido.eat();
        fido.makeNoise();
    }
}
```

In reality, we are decoupling the behaviors from the `Mammal` class. Thus, rather than creating a single `Mammal` entity via inheritance (actually interfaces) we are moving to a composition-based design, similar to the strategy taken in the previous chapter.

In short, by using this approach, we can build `Mammals` with composition rather than being forced to utilize behaviors contained in a single `Mammal` class. For example, suppose someone discovers a `Mammal` that doesn't eat but instead absorbs nutrients through its skin. If we were inheriting from a single `Mammal` class that contains the `eat()` behavior, the new mammal would not need this behavior. However, if we separate all the behaviors into separate, single interfaces, we can build each mammal in exactly the way it presents itself.

5) DIP: Dependency Inversion Principle

The Dependency Inversion Principle states that code should depend on abstractions. It often seems like the terms *dependency inversion* and *dependency injection* are used interchangeably; however, here are some key terms to understand as we discuss this principle:

- **Dependency inversion**—The principle of inverting the dependencies
- **Dependency injection**—The act of inverting the dependencies
- **Constructor injection**—Performing dependency injection via the constructor
- **Parameter injection**—Performing dependency injection via the parameter of a method, like a setter

The goal of dependency inversion is to couple to something abstract rather than concrete.

Although at some point you obviously have to create something concrete, we strive to create a concrete object (by using the `new` keyword) as far up the chain as possible, such as in the `main()` method. Perhaps a better way of thinking of this is to revisit the discussion presented in Chapter 8, "Frameworks and Reuse: Designing with Interfaces and Abstract Classes," where we discuss loading classes at runtime, and in Chapter 9, "Building Objects and Object-Oriented Design," where we talk about decoupling and creating small classes with limited responsibilities.

In the same vein, one of the goals of the Dependency Inversion Principle is to choose objects at runtime, not at compile time. (You can change the behavior of your program at runtime.) You can even write new classes without having to recompile old ones (in fact, you can write new classes and inject them).

Much of the foundation for this discussion was put forth in Chapter 11, "Avoiding Dependencies and Highly Coupled Classes." Let's build on that as we consider the Dependency Inversion Principle.

Step 1: Initial Example

For the first step in this example, we revisit yet again one of the classical object-oriented design examples used throughout this book, that of a `Mammal` class, along with a `Dog` and a `Cat` class that inherit from `Mammal`. The `Mammal` class is abstract and contains a single method called `makeNoise()`.

```
abstract class Mammal
{
    public abstract String makeNoise();
}
```

The subclasses, such as `Cat`, use inheritance to take advantage of `Mammal`'s behavior, `makeNoise()`:

```
class Cat extends Mammal
{
    public String makeNoise()
    {
       return "Meow";
    }
}
```

The main application then instantiates a `Cat` object and invokes the `makeNoise()` method:

```
Mammal cat = new Cat();;

System.out.println("Cat says " + cat.makeNoise());
```

The complete application for the first step is presented in the following code:

Click here to view code image

```
 public class TestMammal {
    public static void main(String args[]) {

        System.out.println("Hello World\n");

        Mammal cat = new Cat();;
        Mammal dog = new Dog();

        System.out.println("Cat says " + cat.makeNoise());
        System.out.println("Dog says " + dog.makeNoise());

    }
}
abstract class Mammal
{
    public abstract String makeNoise();
}
class Cat extends Mammal
{
    public String makeNoise()
    {
        return "Meow";
    }
}
class Dog extends Mammal
{
    public String makeNoise()
    {
        return "Bark";
    }
}
```

Step 2: Separating Out Behavior

The preceding code has a potentially serious flaw: It couples the mammals and the behavior (`makingNoise`). There may be a significant advantage to separating the mammal behaviors from the mammals themselves. To accomplish this, we create a class called `MakingNoise` that can be used by all mammals as well as non-mammals.

In this model, a `Cat`, `Dog`, or `Bird` can then extend the `MakeNoise` class and create their own noise-making behavior specific to their needs, such as the following code fragment for a `Cat`:

```
abstract class MakingNoise
{
        public abstract String makeNoise();
}
class CatNoise extends MakingNoise
{
    public String makeNoise()
    {
        return "Meow";
    }
}
```

With the `MakingNoise` behavior separated from the `Cat` class, we can use the `CatNoise` class in place of the hard coded behavior in the `Cat` class itself, as the following code fragment illustrates:

```
abstract class Mammal
{
    public abstract String makeNoise();
}
class Cat extends Mammal
{
    CatNoise behavior = new CatNoise();
    public String makeNoise()
    {
        return behavior.makeNoise();
    }
}
```

The following is the complete application for the second step:

Click here to view code image

```
public class TestMammal {
    public static void main(String args[]) {

        System.out.println("Hello World\n");

        Mammal cat = new Cat();;
        Mammal dog = new Dog();

        System.out.println("Cat says " + cat.makeNoise());
        System.out.println("Dog says " + dog.makeNoise());

    }
}

abstract class MakingNoise
{
    public abstract String makeNoise();
}
class CatNoise extends MakingNoise
{
    public String makeNoise()
    {
        return "Meow";
    }
}
class DogNoise extends MakingNoise
{
    public String makeNoise()
    {
        return "Bark";
    }
}
abstract class Mammal
{
    public abstract String makeNoise();
}

class Cat extends Mammal
{
    CatNoise behavior = new CatNoise();
    public String makeNoise()
    {
        return behavior.makeNoise();
    }
}
class Dog extends Mammal
{
    DogNoise behavior = new DogNoise();
```

```
    public String makeNoise()
    {
        return behavior.makeNoise();
    }
}
```

The problem is that although we have decoupled a major part of the code, we still haven't reached our goal of dependency inversion because the `Cat` is still instantiating the `Cat` noise-making behavior.

```
CatNoise behavior = new CatNoise();
```

The `Cat` is coupled to the low-level module `CatNoise`. In other words, the `Cat` should not be coupled to `CatNoise` but to the abstraction for making noise. In fact, the `Cat` class should not instantiate its noise-making behavior but instead receive the behavior via injection.

Step 3: Dependency Injection

In this final step, we totally abandon the inheritance aspects of our design and examine how to utilize dependency injection via composition. You do not need inheritance hierarchies, which is one of the major reasons why the concept of composition over inheritance is gaining momentum. You compose a subtype rather than creating a subtype from a hierarchical model.

To illustrate, in the original implementation, the `Cat` and the `Dog` basically contain the same exact code; they simply return a different noise. As a result, a significant percentage of the code is redundant. Thus, if you had many different mammals, there would be a lot of noise-making code. Perhaps a better design is to take the code to make noise out of the mammal.

The major leap here would be to abandon the specific mammals (`Cat` and `Dog`) and simply use the `Mammal` class as shown here:

```
class Mammal
{
    MakingNoise speaker;

    public Mammal(MakingNoisesb)
    {
        this.speaker = sb;
    }
    public String makeNoise()
```

```
    {
        return this.speaker.makeNoise();
    }
}
```

Now we can instantiate a `Cat` noise-making behavior and provide it to the `Animal` class, to make a mammal that behaves like a `Cat`. In fact, you can always assemble a `Cat` by injecting behaviors rather than using the traditional techniques of class building.

```
Mammal cat = new Mammal(new CatNoise());
```

The following is the complete application for the final step:

Click here to view code image

```
public class TestMammal {
    public static void main(String args[]) {

        System.out.println("Hello World\n");

        Mammal cat = new Mammal(new CatNoise());
        Mammal dog = new Mammal(new DogNoise());

        System.out.println("Cat says " + cat.makeNoise());
        System.out.println("Dog says " + dog.makeNoise());

    }
}
class Mammal
{
MakingNoise speaker;

    public Mammal(MakingNoisesb)
    {
        this.speaker = sb;
    }
    public String makeNoise()
    {
        return this.speaker.makeNoise();
    }
}
interface MakingNoise
{
    public String makeNoise();
}
```

```
class CatNoise implements MakingNoise
{
    public String makeNoise()
    {
        return "Meow";
    }
}
class DogNoise implements MakingNoise
{
    public String makeNoise()
    {
        return "Bark";
    }
}
```

When discussing dependency injection, when to actually instantiate an object is now a key consideration. Even though the goal is to compose objects via injection, you obviously must instantiate objects at some point. As a result, the design decisions revolve around when to do this instantiation.

As stated earlier in this chapter, the goal of dependency inversion is to couple to something abstract rather than concrete, even though you obviously must create something concrete at some point. Thus, one simple goal is to create a concrete object (by using `new`) as far up the chain as possible, such as in the `main()` method. Always evaluate things when you see a `new` keyword.

Conclusion

This concludes the discussion of SOLID. The SOLID principles are one of the most influential sets of object-oriented guidelines used today. What is interesting about studying these principles is how they relate to the fundamental object-oriented encapsulation, inheritance, polymorphism, and composition, specifically in the debate of composition over inheritance.

For me, the most interesting point to take away from the SOLID discussion is that nothing is cut and dried. It is obvious from the discussion on composition over inheritance that even the age-old fundamental OO concepts are open for reinterpretation. As we have seen, a bit of time, along with the corresponding evolution in various thought processes, is good for innovation.

References

Martin, Robert, et al. *Agile Software Development, Principles, Patterns, and Practices*. 2002. Boston: Pearson Education, Inc.

Martin, Robert, et al. *Clean Code*. 2009. Boston: Pearson Education, Inc.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q-R

S

T

U

V

W-X-Y-Z

Photo by izusek/gettyimages

Register Your Product at informit.com/register

Access additional benefits and **save 35%** on your next purchase

- Automatically receive a coupon for 35% off your next purchase, valid for 30 days. Look for your code in your InformIT cart or the Manage Codes section of your account page.
- Download available product updates.
- Access bonus material if available.*
- Check the box to hear from us and receive exclusive offers on new editions and related products.

**Registration benefits vary by product. Benefits will be listed on your account page under Registered Products.*

InformIT.com—The Trusted Technology Learning Source

InformIT is the online home of information technology brands at Pearson, the world's foremost education company. At InformIT.com, you can:

- Shop our books, eBooks, software, and video training
- Take advantage of our special offers and promotions (informit.com/promotions)
- Sign up for special offers and content newsletter (informit.com/newsletters)
- Access thousands of free chapters and video lessons

Connect with InformIT—Visit informit.com/community

informIT
the trusted technology learning source

Addison-Wesley • Adobe Press • Cisco Press • Microsoft Press • Pearson IT Certification • Prentice Hall • Que • Sams • Peachpit Press

Pearson

Code Snippets

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a “Click here to view code image” link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

```
public class IntSquare {

    // private attribute
    private int squareValue;

    // public interface
    public in getSquare (int value) {

    SquareValue = calculateSquare(value);

        return squareValue;

    }

    // private implementation
    private in calculateSquare (int value) {

        return value*value;

    }
}
```

```
// private implementation
private intcalculateSquare (int value) {

    return = Math.pow(value,2);

}
```

```
public class Circle extends Shape{

    double radius;

    public Circle(double r) {
        radius = r;
    }

    public double getArea() {
        area = 3.14*(radius*radius);
        return (area);
   }

}
```

```
public class Rectangle extends Shape{

    double length;
    double width;

    public Rectangle(double l, double w){
        length = l;
        width = w;
    }

    public double getArea() {
        area = length*width;
        return (area);
    }

}
```

```
while ( !stack.empty()) {
    Shape shape = (Shape) stack.pop();
    System.out.println ("Area = " + shape.getArea());
}
```

```
public void open(String Name){

        /* Some application-specific processing */

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

        /* Some more application-specific processing */

};
```

```
public void open(String Name){
        /* Some application-specific processing
        /* call the SQLAnywhere API to open the database */
        /* Some more application-specific processing */
};
```

```
public class DataBaseReader {

    String dbName;
    int startPosition;

    // initialize just the name
    public DataBaseReader (String name){
        dbName = name;
        startPosition = 0;
    };

    // initialize the name and the position
    public DataBaseReader (String name, int pos){
        dbName = name;
        startPosition = pos;
    };

    .. // rest of class

}
```

```
try {

    // possible nasty code
    count = 0;
    count = 5/count;

} catch(ArithmeticException e) {

    // code to handle the exception
    System.out.println(e.getMessage());
    count = 1;

}
System.out.println("The exception is handled.");
```

```
public class Number {

    int count;    // available to both method1 and method2

    public method1() {
        count = 1;
    }

    public method2() {
        count = 2;
    }

}
```

```
String firstName = "Joe", lastName = "Smith";

String Name = firstName + " " + lastName;
```

```
/*

  This class defines a cabbie and assigns a cab

*/
```

```
private static String companyName = "Blue Cab Company";
```

```
public Cabbie(String iName, String serialNumber) {

    name = iName;
    myCab = new Cab(serialNumber);

}
```

```
// Get the Name of the Cabbie
public static String getCompanyName() {
    return companyName;
}
```

```
public class Math {

    int temp=0;

    public int swap (int a, int b) {

        temp = a;
        a=b;
        b=temp;

        return temp;

    }

}
```

```
public class Math {

    public int swap (int a, int b) {

        int temp=0;

        temp = a;
        a=b;
        b=temp;

        return temp;

    }

}
```

```
public class DataBaseReader {

    private String db[] = { "Record1","Record2","Record3","Record4","Record5"};
    private booleanDBOpen = false;
    private int pos;

    public void open(String Name){
        DBOpen = true;
    }

    public void close(){
        DBOpen = false;
    }
    public void goToFirst(){
        pos = 0;
    }

    public void goToLast(){
        pos = 4;
    }

    public int howManyRecords(){
        int numOfRecords = 5;
        return numOfRecords;
    }

    public String getRecord(int key){
        /* DB Specific Implementation */
        return db[key];
    }

    public String getNextRecord(){
        /* DB Specific Implementation */
        return db[pos++];
    }

}
```

```
class MainApplication {

    public static void main(String args[]) {

        int x = 0;

        while (x <= 10) {

            if (x==5) System.out.println("x = " + x);
            x++;

        }

    }

}
```

```
public class TestMath {

    public static void main(String[] args) {

        int x = 0;

        SomeMath math = new SomeMath();
        x = math.add(1,2);
        System.out.println("x = " + x);

    }

}
```

```
public class TestBeep {

    public static void main(String[] args) {
        Sound mySound = new Sound();
        mySound.beep();
    }

}
```

```
public abstract class Shape{

    public abstract void draw();

}
public class Circle extends Shape{

    public void draw() {

        System.out.println("I am drawing a Circle");

    }
}

public class Rectangle extends Shape{

    public void draw() {

        System.out.println("I am drawing a Rectangle");

    }
}

public class Star extends Shape{

    public void draw() {

        System.out.println("I am drawing a Star");

    }
}
```

```
public class TestShape {

    public static void main(String args[]) {

        Circle circle = new Circle();
        Rectangle rectangle = new Rectangle();
        Star star = new Star();

        circle.draw();
        rectangle.draw();
        star.draw();

    }

}
```

```
public class Triangle extends Shape{

    public void draw() {

        System.out.println("I am drawing a Triangle");

    }
}
```

```
public class TestShape {

    public static void main(String args[]) {

        Circle circle = new Circle();
        Rectangle rectangle = new Rectangle();
        Star star = new Star();
        Triangle triangle = new Triangle ();

        circle.draw();
        rectangle.draw();
        star.draw();
        triangle.draw();

    }

}

C:\>java TestShape
I am drawing a Circle
I am drawing a Rectangle
I am drawing a Star
I am drawing a Triangle
```

```
public class TestShape {

   public static void main(String args[]) {

     Circle circle = new Circle();
     Rectangle rectangle = new Rectangle();
     Star star = new Star();

     drawMe(circle);
     drawMe(rectangle);
     drawMe(star);

   }

   static void drawMe(Shape s) {
     s.draw();
   }

}
```

```
public abstract class Shape{

   public abstract void draw();

}
```

The Visual Basic .NET code is written like this:

```
Public MustInherit Class Shape

    Public MustOverride Function draw()

End Class
```

```
public class Circle extends Shape {

    public void Draw() {System.out.println ("Draw a Circle")};

}

public class Rectangle extends Shape {

    public void Draw() {System.out.println ("Draw a Rectangle")};

}
```

```
public abstract class Mammal {

    public void generateHeat() {System.out.println("Generate heat");}

    public abstract void makeNoise();

}
```

```
public class Head {

    String size;

    public String getSize() {

        return size;

    }

    public void setSize(String aSize) { size = aSize; }

}
```

```
public class Dog extends Mammal implements Nameable {

    String name;

    Head head;

    public void makeNoise(){System.out.println("Bark");}

    public void setName (String aName) {name = aName;}
    public String getName () {return (name);}

}
```

```
Test.java:6: Incompatible type for Identifier. Can't convert Dog to Head. Head H = D;
```

```
public class Planet {

    String planetName;
    public void getPlanetName() {return planetName;};

}
```

```
public class Car {

    String carName;

    public String getCarName() { return carName; };

}
```

And the Dog class might have code like this:

```
public class Dog {

    String dogName;

    public String getDogName() { return dogName; };

}
```

```
public interface Nameable {

    public String getName();
    public void setName(String aName);

}
```

The new classes, `Planet`, `Car`, and `Dog`, should look like this:

```
public class Planet implements Nameable {

    String planetName;

    public String getName() {return planetName;}
    public void setName(String myName) { planetName = myName; }

}
public class Car implements Nameable {

    String carName;

    public String getName() {return carName;}
    public void setName(String myName) { carName = myName;}

}

public class Dog implements Nameable {

    String dogName;

    public String getName() {return dogName;}
    public void setName(String myName) { dogName = myName;}

}
```

```
public abstract class Shop  {
    CustList customerList;
    public void CalculateSaleTax() {
        System.out.println("Calculate Sales Tax");
    }
    public abstract String[] getInventory();
    public abstract void buyInventory(String item);
}
```

```
public class CustList {

    String name;

    public  String findCust() {return name;}
    public  void addCust(String Name){}

}
```

```
public interface Nameable {

    public abstract String getName();
    public abstract void setName(String name);

}
```

```
public class DonutShop extends Shop implements Nameable {

    String companyName;

    String[] menuItems = {
        "Donuts",
        "Muffins",
        "Danish",
        "Coffee",
        "Tea"
    };

    public String[] getInventory() {

        return menuItems;

    }

    public void buyInventory(String item) {

        System.out.println("\nYou have just purchased " + item);

    }
    public String getName(){

        return companyName;
    }

    public void setName(String name){

        companyName = name;
    }
}
```

```
public class PizzaShop extends Shop implements Nameable {

    String companyName;


    String[] foodOfferings = {
        "Pizza",
        "Spaghetti",
        "Garden Salad",
        "Antipasto",
        "Calzone"
    }

    public String[] getInventory() {

        return foodOfferings;

    }

    public void buyInventory(String item) {

        System.out.println("\nYou have just purchased " + item);

    }

    public String getName(){

        return companyName;
    }

    public void setName(String name){

        companyName = name;
    }

}
```

```
String className = args[0];

Shop  myShop;

myShop = (Shop)Class.forName(className).newInstance();
```

```
class TestShop {

   public static void main (String args[]) {

      Shop shop = null;

      String className = args[0];

      System.out.println("Instantiate the class:" + className + "\n");


      try {

       // new pizzaShop();
         shop = (Shop)Class.forName(className).newInstance();

      } catch (Exception e) {

         e.printStackTrace();
      }

      String[] inventory = shop.getInventory();


      // list the inventory

      for (int i=0; i<inventory.length; i++) {
         System.out.println("Argument" + i + " = " + inventory[i]);
      }


      // buy an item

      shop.buyInventory(Inventory[1]);

   }

}
```

```
import java.util.Date;

public class Employee extends Person{

        private String CompanyID;
        private String Title;
        private Date StartDate;

        private Spouse spouse;
        private Child[] child;
        private Division division;
        private JobDescription[] jobDescriptions;
        public String getCompanyID() {return CompanyID;}
        public String getTitle() {return Title;}
        public Date getStartDate() {return StartDate;}

        public void setCompanyID(String CompanyID) {}
        public void setTitle(String Title) {}
        public void setStartDate(int StartDate) {}

}
```

```
class Circle extends Shape {

    Circle() {
        super(ShapeType.CIRCLE);
        generate();
    }

    @Override
    protected void generate() {
        System.out.println("Generating a Circle");
    }
}

class Square extends Shape {

    Square() {
        super(ShapeType.SQUARE);
        generate();
    }
    @Override
    protected void generate() {
        System.out.println("Generating a Square");
    }
}

class Triangle extends Shape {

    Triangle() {
        super(ShapeType.TRIANGLE);
        generate();
    }

    @Override
    protected void generate() {
        System.out.println("Generating a Triangle");
    }
```

```
}
```

```
class ShapeFactory {
    public static Shape generateShape(ShapeType sType) {
        Shape shape = null;
        switch (sType) {

        case CIRCLE:
            shape = new Circle();
            break;

        case SQUARE:
            shape = new Square();
            break;

        case TRIANGLE:
            shape = new Triangle();
            break;

        default:
            // throw an exception
            break;
        }
        return shape;
    }
}
```

```
public class TestFactoryPattern {
    public static void main(String[] args) {

        Circle circle = new Circle();
        Square square = new Square();
        Triangle triangle = new Triangle();

    }
}
```

```
public class TestFactoryPattern {
    public static void main(String[] args) {

        ShapeFactory.generateShape(ShapeType.CIRCLE);
        ShapeFactory.generateShape(ShapeType.SQUARE);
        ShapeFactory.generateShape(ShapeType.TRIANGLE);

    }
}
```

```
package MailTool;
public class MailTool {
   public MailTool () {
   }
   public int retrieveMail() {

    System.out.println ("You've Got Mail");

         return 0;
   }
}
```

```
package MailTool;
class MyMailTool implements MailInterface {
   private MailTool yourMailTool;
   public MyMailTool () {
    yourMailTool= new MailTool();
        setYourMailTool(yourMailTool);
   }
   public int getMail() {
      return getYourMailTool().retrieveMail();
   }
   public MailTool getYourMailTool() {
      return yourMailTool ;
   }
   public void setYourMailTool(MailTool newYourM
      yourMailTool = newYourMailTool;
   }
}
```

```
package MailTool;
public class Adapter
{
  public static void main(String[] args)
  {
    MyMailTool myMailTool = new MyMailTool();

    myMailTool.getMail();

  }
}
```

```
package Iterator;

import java.util.*;
public class Iterator {
    public static void main(String args[]) {

        // Instantiate an ArrayList.
        ArrayList<String> names = new ArrayList();

        // Add values to the ArrayList
        names.add(new String("Joe"));
        names.add(new String("Mary"));
        names.add(new String("Bob"));
        names.add(new String("Sue"));

        //Now Iterate through the names
        System.out.println("Names:");
        iterate(names);
    }

    private static void iterate(ArrayList<String> arl) {
        for(String listItem : arl) {
                    System.out.println(listItem.toString());
                }
    }
}
```

```
class Mammal {
    public void eat () {System.out.println("I am Eating");};
}
class Bat extends Mammal {
    public void fly () {System.out.println("I am Flying");};
}
class Dog extends Mammal {
    public void walk () {System.out.println("I am Walking");};
}
public class TestMammal {

    public static void main(String args[]) {

        System.out.println("Composition over Inheritance");;

        System.out.println("\nDog");
        Dog fido = new Dog();
        fido.eat();
        fido.walk();
        System.out.println("\nBat");
        Bat brown = new Bat();
        brown.eat();
        brown.fly();

    }

}
```

```
class Mammal {
    public void eat () {System.out.println("I am Eating");};
}
class Walkable {
    public void walk () {System.out.println("I am Walking");};
}
class Flyable {
    public void fly () {System.out.println("I am Flying");};
}
class Dog {
    Mammal dog = new Mammal();
    Walkable walker = new Walkable();
}
class Bat {
    Mammal bat = new Mammal();
    Flyable flyer = new Flyable();
}

public class TestMammal {

    public static void main(String args[]) {

        System.out.println("Composition over Inheritance");;
        System.out.println("\nDog");;
        Dog fido = new Dog();
        fido.dog.eat();
        fido.walker.walk();

        System.out.println("\nBat");;
        Bat brown = new Bat();
        brown.bat.eat();
        brown.flyer.fly();

    }
```

```
}
```

```
class Swimmable {
    public void fly () {System.out.println("I am Swimming");};
}
class Whale {
    Mammal whale = new Mammal();
    Walkable swimmer = new Swimmable ();
}
```

```
class Dog extends Mammal implements IWalkable{
    Walkable walker;
    public void setWalker (Walkable w) {
        this.walker=w;
    }
    public void walk () {System.out.println("I am Walking");};
}
```

```
class Mammal {
    public void eat () {System.out.println("I am Eating");};
}
interface IWalkable {
    public void walk();
}
class Dog extends Mammal implements IWalkable{
    Walkable walker;
    public void setWalker (Walkable w) {
        this.walker=w;
    }
    public void walk () {System.out.println("I am Walking");};
}

public class TestMammal {

    public static void main(String args[]) {

        System.out.println("Composition over Inheritance");
        System.out.println("\nDog");
        Walkable walker = new Walkable();
        Dog fido = new Dog();
        fido.setWalker(walker);
        fido.eat();
        fido.walker.walk();

    }

}
```

```
class CalculateAreas {
    Shape[] shapes;
    double sumTotal=0;
    public CalculateAreas(Shape[] sh){
        this.shapes = sh;
    }
    public double sumAreas() {
        sumTotal=0;
        for (inti=0; i<shapes.length; i++) {
            sumTotal = sumTotal + shapes[i].calcArea();
        }
        return sumTotal;
    }
    public void output() {
        System.out.println("Total of all areas = " + sumTotal);
    }
}
```

```
public class TestShape {
    public static void main(String args[]) {

        System.out.println("Hello World!");

        Circle circle = new Circle(1);

        Shape[] shapeArray = new Shape[1];
        shapeArray[0] = circle;

        CalculateAreas ca = new CalculateAreas(shapeArray);

        ca.sumAreas();
        ca.output();
    }
}
```

```
class CalculateAreas {
    Shape[] shapes;
    double sumTotal=0;

    public CalculateAreas(Shape[] sh){
        this.shapes = sh;
    }

    public double sumAreas() {
        sumTotal=0;

        for (inti=0; i<shapes.length; i++) {

            sumTotal = sumTotal + shapes[i].calcArea();

        }
            return sumTotal;
        }
    }
    class OutputAreas {
        double areas=0;
        public OutputAreas(double a){
            this.areas = a;
        }
        public void console() {
            System.out.println("Total of all areas = " + areas);
        }

        public void HTML() {
            System.out.println("<HTML>");
            System.out.println("Total of all areas = " + areas);
            System.out.println("</HTML>");
        }
    }
```

```
public class TestShape {
    public static void main(String args[]) {

        System.out.println("Hello World!");

        Circle circle = new Circle(1);

        Shape[] shapeArray = new Shape[1];
        shapeArray[0] = circle;

        CalculateAreas ca = new CalculateAreas(shapeArray);

        CalculateAreas sum = new CalculateAreas(shapeArray);
        OutputAreasoAreas = new OutputAreas(sum.sumAreas());

        oAreas.console();    // output to console
        oAreas.HTML();       // output to HTML

    }
}
```

```
class Rectangle{
    protected double length;
    protected double width;

    public Rectangle(double l, double w) {
        length = l;
        width = w;
    };
}
class CalculateAreas {

    private double area;

    public double calcArea(Rectangle r) {

        area = r.length * r.width;

        return area;

    }
}
public class OpenClosed {
    public static void main(String args[]) {

        System.out.println("Hello World");

        Rectangle r = new Rectangle(1,2);

        CalculateAreas ca = new CalculateAreas ();

        System.out.println("Area = "+ ca.calcArea(r));

    }
```

```
}
```

```
abstract class Shape {
    public abstract double getArea();
}
class Rectangle extends Shape
{
    protected double length;
    protected double width;

    public Rectangle(double l, double w) {
        length = l;
        width = w;
    };
    public double getArea() {
        return length*width;
    }

}
class Circle extends Shape
{
    protected double radius;

    public Circle(double r) {
        radius = r;
    };
    public double getArea() {
        return radius*radius*3.14;
    }
}
class CalculateAreas {
    private double area;

    public double calcArea(Shape s) {
        area = s.getArea();
        return area;
    }
```

```
}
```

```
public class OpenClosed {
    public static void main(String args[]) {

        System.out.println("Hello World");

        CalculateAreas ca = new CalculateAreas();

        Rectangle r = new Rectangle(1,2);

        System.out.println("Area = " + ca.calcArea(r));

        Circle c = new Circle(3);

        System.out.println("Area = " + ca.calcArea(c));

    }
}
```

```
abstract class Shape{
    protected double area;

    public abstract double calcArea();
}
class Rectangle extends Shape{
    private double length;
    private double width;
    public Rectangle(double l, double w){
        length = l;
        width = w;
    }
    public double calcArea() {
        area = length*width;
        return (area);
    };

}
class Square extends Rectangle{
    public Square(double s){
        super(s, s);
    }
}

public class LiskovSubstitution {
    public static void main(String args[]) {

        System.out.println("Hello World");

        Rectangle r = new Rectangle(1,2);

        System.out.println("Area = " + r.calcArea());

        Square s = new Square(2);

        System.out.println("Area = " + s.calcArea());

    }
```

```
}
```

```
 abstract class Shape {
    protected double area;

    public abstract double calcArea();
}

class Rectangle extends Shape {

    private double length;
    private double width;

    public Rectangle(double l, double w) {
        length = l;
        width = w;
    }

    public double calcArea() {
        area = length*width;
        return (area);
    };
}

class Square extends Shape {
    private double side;

    public Square(double s){
        side = s;
    }
    public double calcArea() {
        area = side*side;
        return (area);
    };
```

```
}
```

```
public class LiskovSubstitution {
    public static void main(String args[]) {

        System.out.println("Hello World");

        Rectangle r = new Rectangle(1,2);

        System.out.println("Area = " + r.calcArea());

        Square s = new Square(2);

        System.out.println("Area = " + s.calcArea());

    }
}
```

```
interface IMammal {
    public void eat();
    public void makeNoise();
}
class Dog implements IMammal {
    public void eat() {
        System.out.println("Dog is eating");
    }
    public void makeNoise() {
        System.out.println("Dog is making noise");
    }
}
public class MyClass {
    public static void main(String args[]) {

        System.out.println("Hello World");

        Dog fido = new Dog();
        fido.eat();
        fido.makeNoise();
    }
}
```

```
interface IEat {
    public void eat();
}
interface IMakeNoise {
    public void makeNoise();
}
class Dog implements IEat, IMakeNoise {
    public void eat() {
        System.out.println("Dog is eating");
    }
    public void makeNoise() {
        System.out.println("Dog is making noise");
    }
}
public class MyClass {
    public static void main(String args[]) {

        System.out.println("Hello World");

        Dog fido = new Dog();
        fido.eat();
        fido.makeNoise();
    }
}
```

```
 public class TestMammal {
    public static void main(String args[]) {

        System.out.println("Hello World\n");

        Mammal cat = new Cat();;
        Mammal dog = new Dog();

        System.out.println("Cat says " + cat.makeNoise());
        System.out.println("Dog says " + dog.makeNoise());

    }
}
abstract class Mammal
{
    public abstract String makeNoise();
}
class Cat extends Mammal
{
    public String makeNoise()
    {
        return "Meow";
    }
}
class Dog extends Mammal
{
    public String makeNoise()
    {
        return "Bark";
    }
}
```

```
public class TestMammal {
    public static void main(String args[]) {

        System.out.println("Hello World\n");

        Mammal cat = new Cat();;
        Mammal dog = new Dog();

        System.out.println("Cat says " + cat.makeNoise());
        System.out.println("Dog says " + dog.makeNoise());

    }
}

abstract class MakingNoise
{
    public abstract String makeNoise();
}
class CatNoise extends MakingNoise
{
    public String makeNoise()
    {
        return "Meow";
    }
}
class DogNoise extends MakingNoise
{
    public String makeNoise()
    {
        return "Bark";
    }
}
abstract class Mammal
{
    public abstract String makeNoise();
}

class Cat extends Mammal
{
    CatNoise behavior = new CatNoise();
    public String makeNoise()
    {
        return behavior.makeNoise();
    }
}
class Dog extends Mammal
{
    DogNoise behavior = new DogNoise();
    public String makeNoise()
    {
        return behavior.makeNoise();
    }
```

}

```
public class TestMammal {
    public static void main(String args[]) {

        System.out.println("Hello World\n");

        Mammal cat = new Mammal(new CatNoise());
        Mammal dog = new Mammal(new DogNoise());

        System.out.println("Cat says " + cat.makeNoise());
        System.out.println("Dog says " + dog.makeNoise());

    }
}
class Mammal
{
MakingNoise speaker;

    public Mammal(MakingNoisesb)
    {
        this.speaker = sb;
    }
    public String makeNoise()
    {
        return this.speaker.makeNoise();
    }
}
```

```
interface MakingNoise
{
    public String makeNoise();
}
class CatNoise implements MakingNoise
{
    public String makeNoise()
    {
        return "Meow";
    }
}
class DogNoise implements MakingNoise
{
    public String makeNoise()
    {
        return "Bark";
    }
}
```