Table of Contents

About the Author

Matt Weisfeld is an associate professor in business & technology at Cuyahoga Community College (Tri-C) in Cleveland, Ohio. A member of the Information Technology faculty, he focuses on programming, web development, and entrepreneurship. Prior to joining Tri-C, Weisfeld spent 20 years in the information technology industry gaining experience in software development, project management, small business management, corporate training, and part-time teaching. He holds an MS in computer science and an MBA in project management. Besides the first two editions of *The Object-Oriented Thought Process*, he has published two other computer books and articles in magazines and journals such as *developer.com*, *Dr. Dobb's Journal*, *The C/C++ Users Journal*, *Software Development Magazine*, *Java Report*, and the international journal *Project Management*.

Dedication

To Sharon, Stacy, Stephanie, and Duffy

Acknowledgments

As with the first two 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 of her help. Not only did she provide support and encouragement throughout this lengthy process, she is also the first line editor for all of my writing.

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

I have really enjoyed working with the people at Pearson on all three editions of this book. Working with editors Mark Taber, Seth Kerney, Vanessa Evans, and Songlin Qiu has been a pleasure.

A special thanks goes to Jon Upchurch for his help with much of the code as well as the technical editing of the manuscript.

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

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.

You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

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

Email: feedback@developers-library.info

Mail: Mark Taber
Associate Publisher
Pearson Education
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

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

Introduction

This Book's Scope

As the title indicates, this book is about the object-oriented (OO) thought process. Obviously, choosing the theme and title of the book are important decisions; however, these decisions were not all that simple. Numerous books deal with various levels of object orientation. Several popular books deal with topics including OO analysis, OO design, OO programming, design patterns, OO data (XML), the Unified Modeling Language (UML), OO Internet development, various OO programming languages, and many other topics related to OO development.

However, while pouring over all of these books, many people forget that all of these topics are built on a single foundation: how you think in OO ways. It is unfortunate, but software professionals often dive into these books without taking the appropriate time and effort to *really* understand the concepts behind the content.

I contend that learning OO concepts is not accomplished by learning a specific development method or a set of tools. Doing things in an OO manner is, simply put, a way of thinking. This book is all about the OO thought process.

Separating the methods and tools from the OO thought process is not easy. Many people are introduced to OO concepts via one of these methods or tools. For example, years ago, most C programmers were first introduced to object orientation by migrating directly to C++—before they were even remotely exposed to OO concepts. Other software professionals were first introduced to object orientation by presentations that included object models using UML—again, before they were even exposed directly to OO concepts. It is not unusual to find that programming books and courses defer OO concepts until later in the learning process.

It is important to understand the significant difference between learning OO concepts and using the methods and tools that support the paradigm. This came into focus for me before I worked on the first edition of this book when I read articles such as 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.

Although learning a modeling language is an important step, it is much more important to learn OO skills first. Learning UML before 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. Many times developers who claim to be C++ programmers are simply C programmers using C++ compilers. Even now, with languages such as C# .NET, VB .NET, 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 (writing only C syntax) while forsaking all of C++'s OO features. 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 on the road to OO development, you first learn the fundamental OO concepts. Resist the temptation to jump directly into a programming language (such as VB .NET, C++, C# .NET or Java) or a modeling language (such as UML), and take the time to learn the object-oriented thought process.

In my first class in Smalltalk in the late 1980s, the instructor told the class that the new OO paradigm was a totally new way of thinking (*despite the fact that it has been around since the 60s*). He went on to say that although all of us were most likely very good programmers, about 10%–20% of us would never really grasp the OO way of doing things. If this statement is indeed true, it is most likely because some people never really take the time to make the paradigm shift and learn the underlying OO concepts.

What's New in the Third Edition

As stated often in this introduction, my vision for the first edition was primarily a conceptual book. Although I still adhere to this goal for the second and third editions, I have included several application topics that fit well with object-oriented concepts. For the third edition I expand on many of the topics of the second edition and well as include totally new chapters. These revised and updated concepts

- XML is used for object communication.
- Object persistence and serialization.
- XML integrated into the languages object definition.
- Adding properties to attributes.
- XML-based Internet applications.
- Client/Server technologies.
- Expanded code examples in Java, C# .NET and VB .NET.

The chapters that cover these topics are still conceptual in nature; however, many of the chapters include Java code that shows how these concepts are implemented. In this third edition, a code appendix is included that presents the chapter's examples in C# .NET and Visual Basic .NET.

The Intended Audience

This book is a general introduction to fundamental OO concepts with code examples to reinforce the concepts. One of the most difficult juggling acts was to keep the material conceptual while still providing a solid, technical code base. The goal of this book is to allow a reader to understand the concepts and technology without having a compiler at hand. However, if you do have a compiler available, then there is code to be investigated.

The intended audience includes business managers, designers, developers, programmers, project managers, and anyone who wants to gain a general understanding of what object orientation is all about. Reading this book should provide a strong foundation for moving to other books covering more advanced OO topics.

Of these more advanced books, one of my favorites remains *Object-Oriented Design in Java* by Stephen Gilbert and Bill McCarty. I really like the approach of the book, and have used it as a textbook in classes I have taught on OO concepts. I cite *Object-Oriented Design in Java* often throughout this book, and I recommend that you graduate to it after you complete this one.

Other books that I have found very helpful include *Effective C++* by Scott Meyers, *Classical and Object-Oriented Software Engineering* by Stephen R. Schach, *Thinking in C++* by Bruce Eckel, *UML Distilled* by Martin Flower, and *Java Design* by Peter Coad and Mark Mayfield.

The conceptual nature of this book provides a unique perspective in regards to other computer technology books. While books that focus on specific technologies, such as programming languages, struggle with the pace of change, this book has the luxury of presenting established concepts that, while certainly being fine-tuned, do not experience radical changes. With this in mind, many of the books that were referenced several years ago, are still referenced because the concepts are still fundamentally the same.

This Book's Scope

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 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 Java, C# .NET, and VB .NET code and so many UML diagrams? First, they are all great for illustrating OO concepts. Second, both are vital to the OO process and should be addressed at an introductory level. The key is not to focus on Java, C# .NET, and VB .NET or UML, but to use them as aids in the understanding of the underlying concepts.

The Java, C# .NET and VB .NET 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, and VB .NET or UML, all of which can command volumes unto themselves. 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.

This Book's Conventions

The following conventions are used in this book:

- Code lines, commands, statements, and any other code-related terms appear in a `monospace` typeface.
- Placeholders that stand for what you should actually type appear in *`italic monospace`*. Text that you should type appears in **`bold monospace`**.
- Throughout the book, there are special sidebar elements, such as

> **Note**
>
> A Note presents interesting information related to the discussion—a little more insight or a pointer to some new technique.

> **Tip**
>
> A Tip offers advice or shows you an easier way of doing something.

> **Caution**
>
> A Caution alerts you to a possible problem and gives you advice on how to avoid it.

Source Code Used in This Book

You can download all the source code and examples discussed within this book from the publisher's website.

1

Introduction to Object-Oriented Concepts

Although many people find this bit of information surprising, object-oriented (OO) software development has been around since the early 1960s. Objects are now used throughout the software development industry. It is no secret that the software industry can be slow-moving at times. It is also true that, when systems are working fine, there has to be a compelling reason to replace them. This has somewhat slowed the propagation of OO systems. There are many non-OO *legacy systems* (that is, older systems that are already in place) that are doing the job—so why risk potential disaster by changing them? In most cases you should not change them, at least not simply 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.

Although there has been a steady and significant growth in OO development in the past 15 years, the continued reliance on the Internet has helped catapult it even further into the mainstream. The emergence of day-to-day business transactions on the Internet has opened a brand-new arena, 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 a structured module 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, non-portable hardware features, and so on.

Today, one of the most interesting areas of software development is the marriage of legacy and Internet based systems. In many cases, a web-based front-end ultimately connects to data that resides on a Mainframe. Developers who can combine the skills of mainframe and web development are in demand.

Objects have certainly made their way into our personal and professional information systems (IS) lives—and they cannot be ignored. You probably experience objects in your daily like without even knowing it. These experiences can take place in your car, talking on your cell phone, using your digital TV, and many other situations.

With the success of Java, Microsoft's .NET technologies and many others, objects are becoming a major part of the technology equation. With the explosion of the Internet, and countless local networks, the electronic highway has in essence become an object-based highway (in the case of wireless, object-based signals). As businesses gravitate toward the Web, they are gravitating toward objects because the technologies used for electronic commerce are mostly OO in nature.

This chapter is an overview of the fundamental OO concepts. 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.

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.

For example, when you look at a person, you see the person as an object. And an object is defined by two terms: 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.

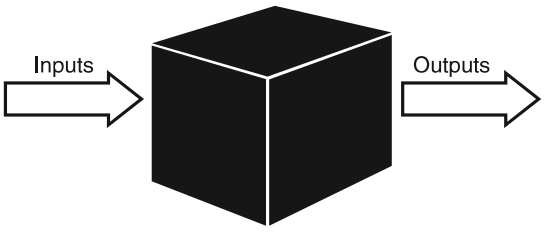


Figure 1.1 Black boxes.

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 slowed its acceptance was the fact that there were a lot of non-OO systems in place that worked perfectly fine. Thus, it did not make any business sense to simply 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 acceptance of OO development it seemed somewhat likely that OO databases would replace relational databases. However, this never happened. Businesses had a lot of money invested in relational databases, and there was one overriding factor—they worked. When all of 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 sometimes 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.

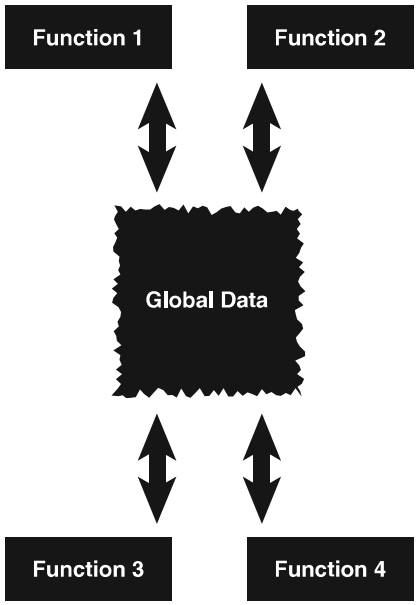


Figure 1.2 Using global data.

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 importantly, 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 is 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 for 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.

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

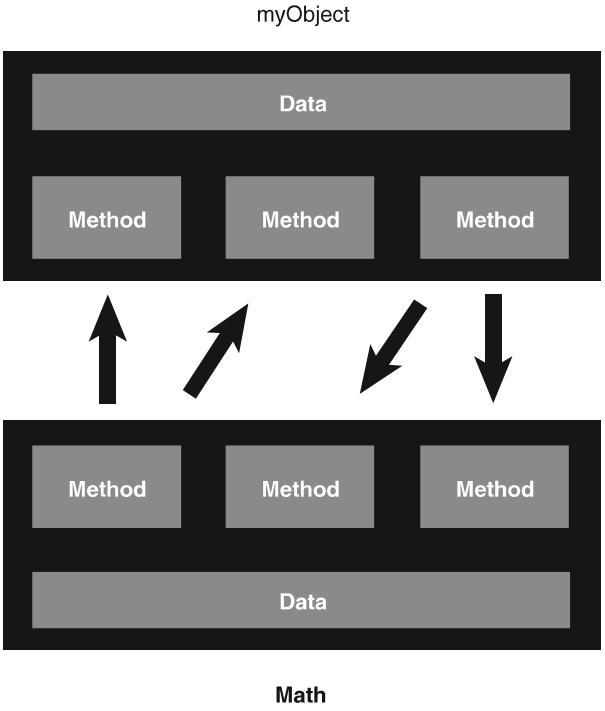


Figure 1.3 Object-to-object communication.

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 proper 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 about 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 server to transmit the data. In this model, it is possible that no code is actually sent over the wire.

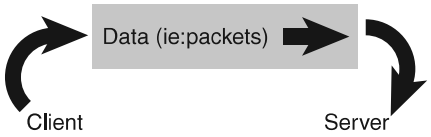


Figure 1.4 Data transmitted over a wire.

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

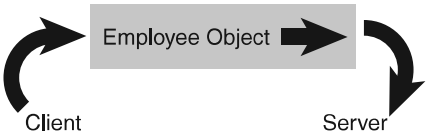


Figure 1.5 Objects transmitted over a wire.

> **Proper Design**
>
> A good example of this concept is a Web object, such as a Java object/applet. The browser has no idea of what the Web object will do—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.

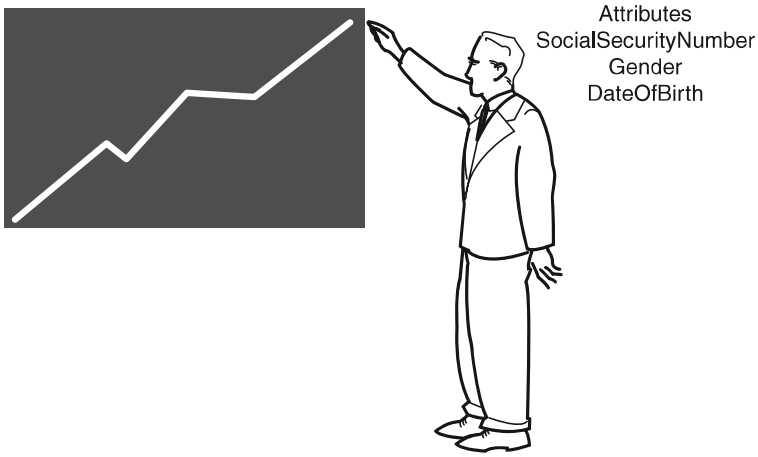


Figure 1.6 Employee attributes.

Object Behaviors

The *behavior* of an object is 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 in comes to data. As we will see in Chapter 11, *Objects and Portable Data: XML* and Chapter 12, *Persistent Objects: Serialization and Relational Databases*, data is now constructed in an object-oriented manner. 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 actually 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:

```
[XmlAttribute("name")]
 public String Name
 {
     get
     {
         return this.strName;
     }
     set
     {
         if (value == null) return;
         this.strName = value;
     }
 }
```

In this approach, 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 when we cover object-oriented data in Chapter 11 and beyond.

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 only showing 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 further illustrate behaviors, consider Figure 1.7. 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

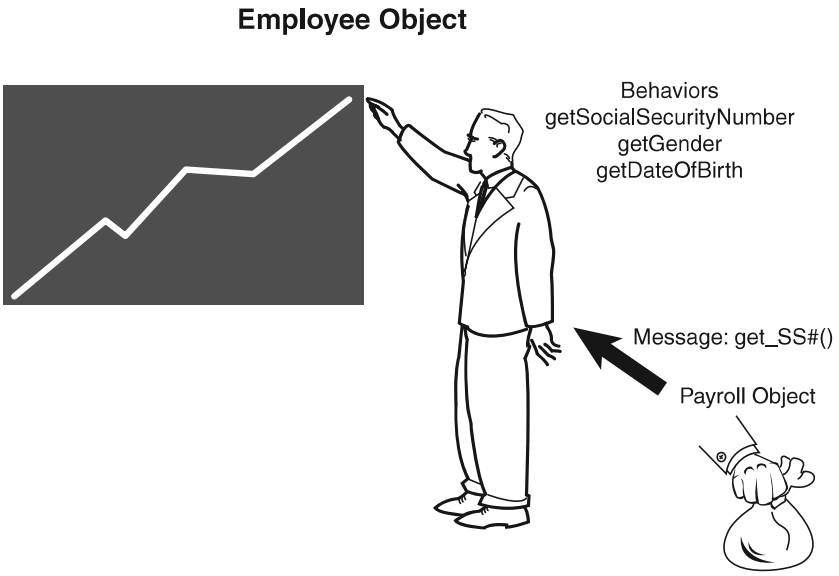


Figure 1.7 Employee behaviors.

`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.

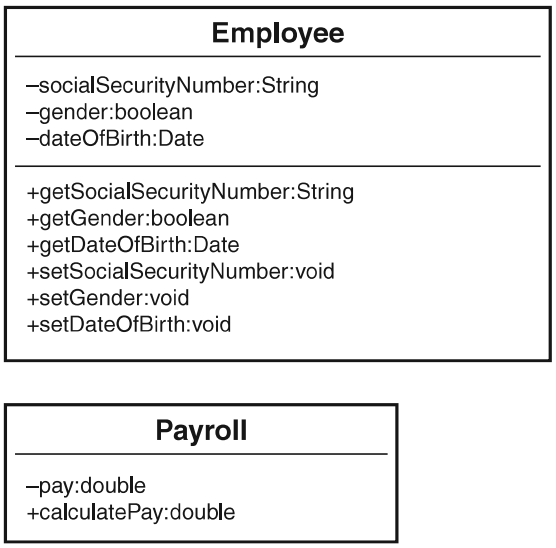


Figure 1.8 Employee and payroll class diagrams.

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, "Advanced 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`, while 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). UML is discussed throughout this book, and you can find a description of this notation in Chapter 10, "Creating Object Models with UML."

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.

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 visa 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.

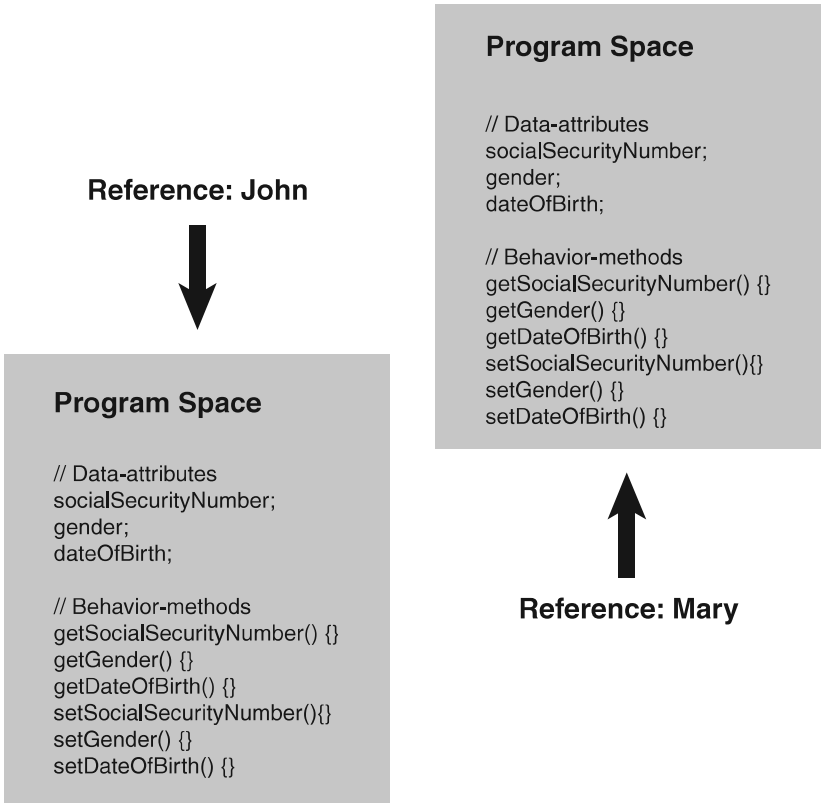


Figure 1.9 Program spaces.

To explain classes and methods, it's helpful to use an example from the relational database world. In a database table, the definition of the table itself (fields, description, and data types used) would be a class (metadata), and the objects would be the rows of the table (data).

This book focuses on the concepts of OO software and not on a specific implementation (such as Java, C#,Visual Basic .NET, or C++), but 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, the end of each chapter will contain the example code in C# .NET,VB .NET, and C++ as well (when applicable).

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

Classes Are Object Templates

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.

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;
```

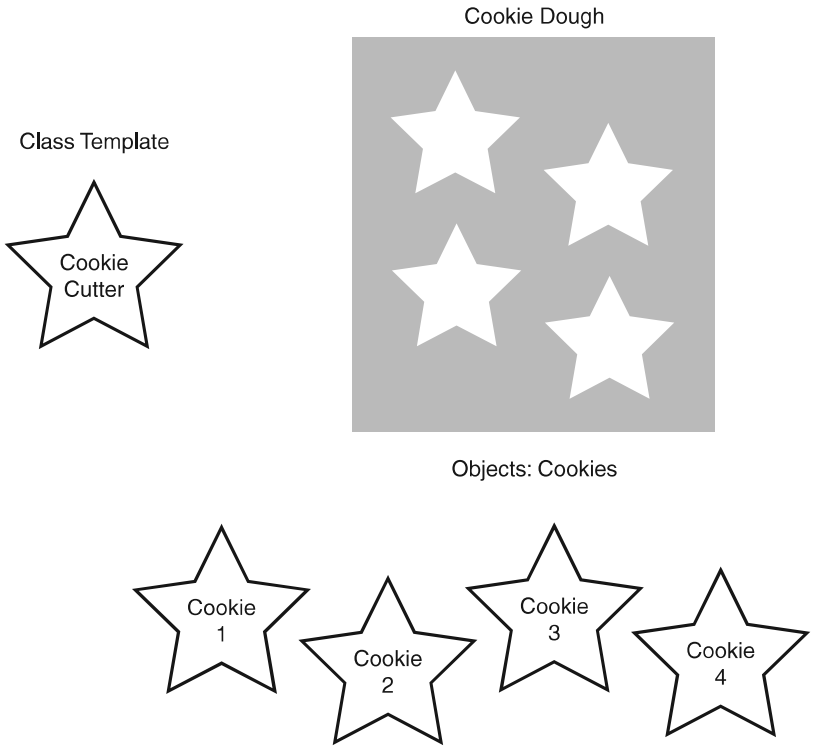


Figure 1.10 Class template.

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

```
myClass myObject;
```

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 (analogous to fields) and behaviors (analogous to 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 setName(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, "Advanced Object-Oriented Concepts."

Methods

As you learned earlier in the chapter, methods implement the required behavior of a class. Every object instantiated from this class has the 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 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();

    String = p.setName("Joe");

    ... code

    String = 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, as we are really interested in the concepts. We address the code in detail as we progress through the book.

Using UML to Model a Class Diagram

Over the years, many tools and modeling methodologies have been developed to assist in designing software systems. One of the most popular tools in use today is Unified Modeling Language (UML). Although it is beyond the scope of this book to describe UML in fine 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.

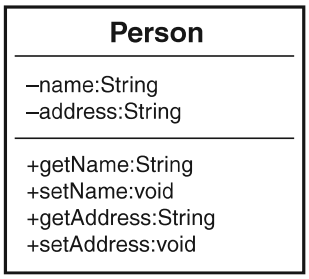


Figure 1.11 The **`Person`** class diagram.

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 only reveal 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.

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, 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 only be changed in one place (the setter).

From a security perspective, you don't want uncontrolled code to change or retrieve data such as passwords and personal information.

Interfaces Versus Interfaces

It is important to note that there are interfaces to the classes as well as the methods—don't confuse the two. The interfaces to the classes are the public methods while the interfaces to the methods relate to how you call (invoke) them. This will be covered in more detail later.

Implementations

Only the public attributes and methods are considered the interface. The user should not see any part of the implementation—interacting with an object solely through class interfaces. 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 use a cord that complies with the electrical outlet specifications; this is the interface between the toaster and the power company (actually the power industry). The fact 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 seen in Figure 1.12.

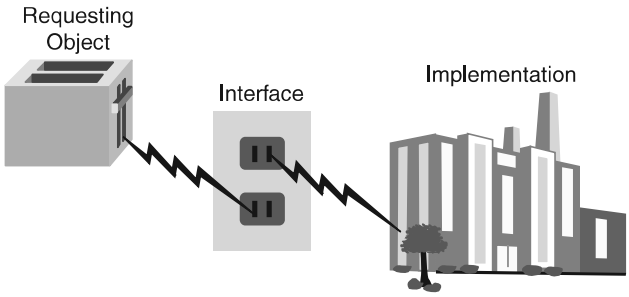


Figure 1.12 Power plant example.

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 provide 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.

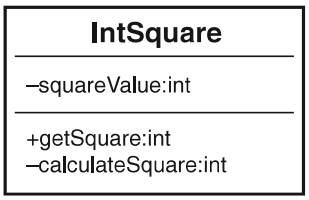


Figure 1.13 The square class.

This class diagram corresponds to the following code:

```
public class IntSquare {

    // private attribute
    private int squareValue;

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

        SquareValue =calculateSquare(value);

        return squareValue;

    }

    // private implementation
    private int 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 only reveals 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—say, you wanted to use 'the language's built-in square function—you would not need to change the interface. The user would get the same functionality, 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

One of the most powerful features of OO programming is, perhaps, code reuse. Structured design provides code reuse to a certain extent—you can write a procedure and then use it as many times as you want. However, OO design goes an important step further, allowing you to define relationships between classes that facilitate not only code reuse, but also better overall design, by organizing classes and factoring in commonalties of various classes. *Inheritance* is a primary means of providing this functionality.

Inheritance allows a class to inherit the attributes and methods of another class. This allows creation of brand new classes by abstracting out common attributes and behaviors.

One of the major design issues in OO programming is to factor out commonality of the various classes. For example, say 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.

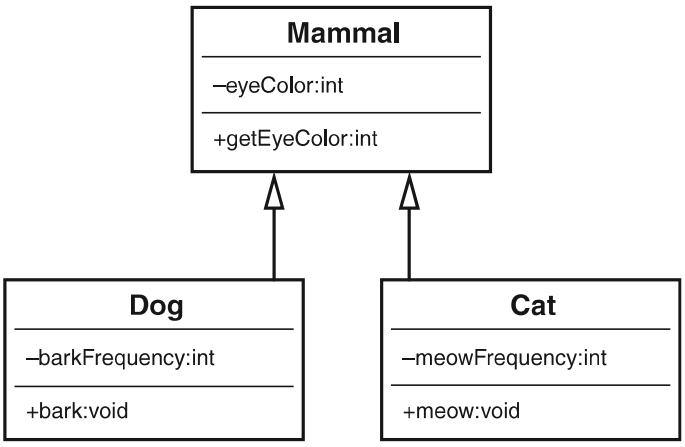


Figure 1.14 Mammal hierarchy.

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

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

In the same vein, `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.

Superclasses and Subclasses

The superclass, or parent 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. 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.

In most recent OO languages (such as Java and .NET), a class can only have 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.

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).

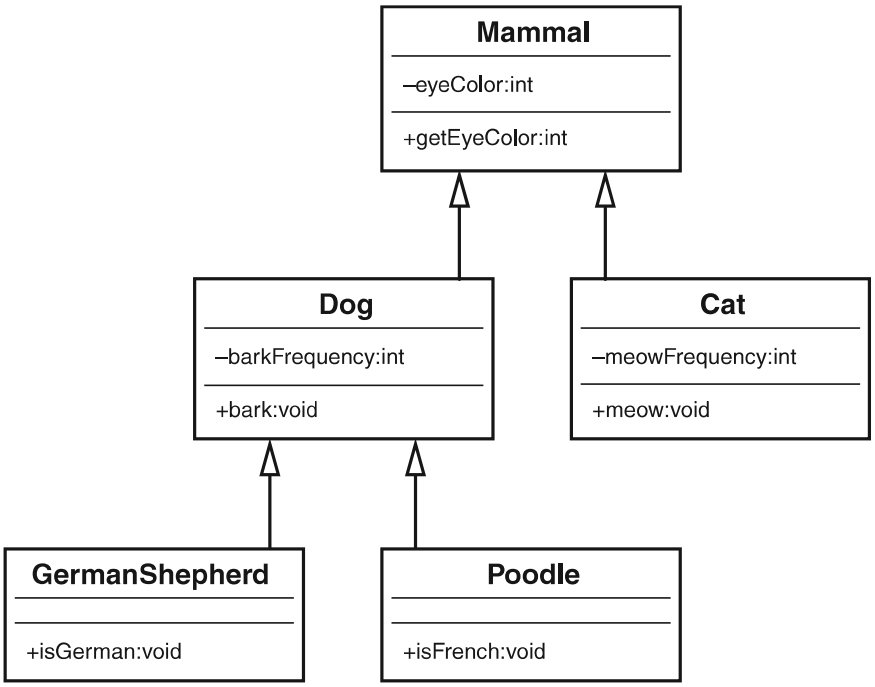


Figure 1.15 Mammal UML diagram.

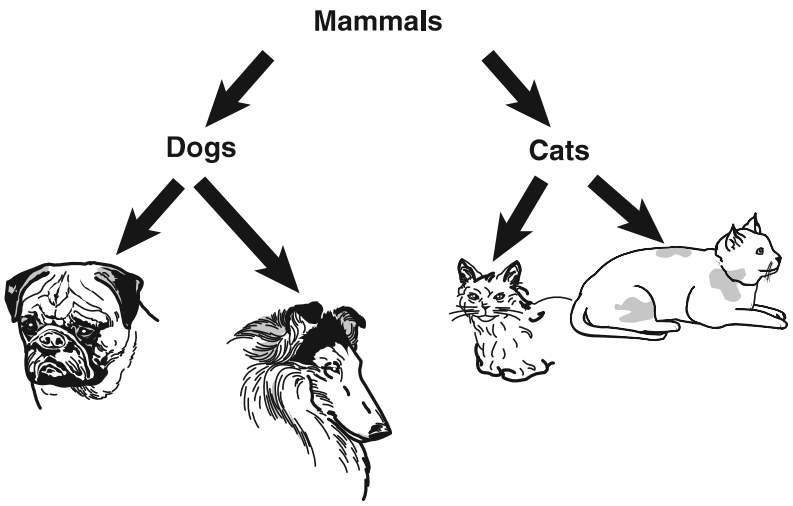


Figure 1.16 Mammal hierarchy.

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 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 like drawing and word processing applications.

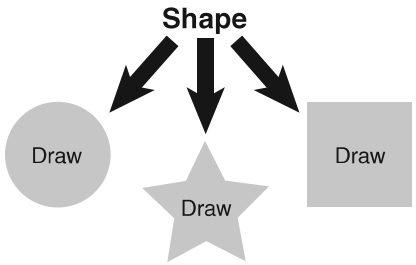


Figure 1.17 The shape hierarchy.

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, as 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`):

```
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`:

```
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.

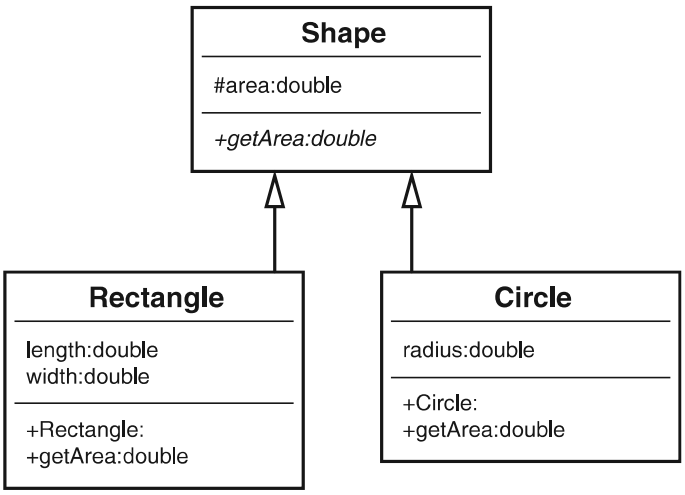


Figure 1.18 Shape UML diagram.

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 simply 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):

```
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` will calculate the area for a circle, and `Rectangle` will calculate 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 across classes, as well as applications. Consider an office suite that includes a word processing and a spreadsheet application. Let's assume that both have a method called `Print`. This `Print` method can be part of the `Office` class as a requirement any class that inherits from it to implement a `Print` method. The interesting thing here is that although both the word processor and spreadsheet do different things when the `Print` method is invoked, one prints a processing document and the other a spreadsheet document.

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

Although an inheritance relationship is considered an *is-a* relationship for reasons already discussed, 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 is 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 polymor-

phism, 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.

Example Code Used in This Chapter

The following code is presented in C# .NET and VB .NET. These examples correspond to the Java code that is listed inside the chapter itself.

The `TestPerson` Example: C# .NET

```
using System;

namespace ConsoleApplication1
{
    class TestPerson
    {
        static void Main(string[] args)
        {

            Person joe = new Person();

            joe.Name = "joe";

            Console.WriteLine(joe.Name);

            Console.ReadLine();
        }
    }

    public class Person
    {

        //Attributes
        private String strName;
        private String strAddress;

        //Methods
        public String Name
        {
            get { return strName; }
            set { strName = value; }
        }
```

```
        public String Address
        {
            get { return strAddress; }
            set { strAddress = value; }
        }

    }
}
```

The `TestPerson` Example: VB .NET

```
Module TestPerson

    Sub Main()

        Dim joe As Person = New Person

        joe.Name = “joe”

        Console.WriteLine(joe.Name)

        Console.ReadLine()

    End Sub

End Module

Public Class Person

    Private strName As String
    Private strAddress As String

    Public Property Name() As String
        Get
            Return strName
        End Get
        Set(ByVal value As String)
            strName = value
        End Set
    End Property

    Public Property Address() As String
        Get
            Return strAddress
        End Get
```

```
        Set(ByVal value As String)
            strAddress = value
        End Set
    End Property

End Class
```

The `TestShape` Example: C# .NET

```
using System;

namespace TestShape
{
    class TestShape
    {
        public static void Main()
        {

            Circle circle = new Circle(5);
            Console.WriteLine(circle.calcArea());

            Rectangle rectangle = new Rectangle(4, 5);
            Console.WriteLine(rectangle.calcArea());

            Console.ReadLine();

        }
    }
    public abstract class Shape
    {

        protected double area;

        public abstract double calcArea();

    }
    public class Circle : Shape
    {

        private double radius;

        public Circle(double r)
        {

            radius = r;
```

```
        }

        public override double calcArea()
        {

            area = 3.14 * (radius * radius);
            return (area);

        }
    }
    public class Rectangle : Shape
    {

        private double length;
        private double width;

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

        public override double calcArea()
        {
            area = length * width;
            return (area);
        }
    }
}
```

The `TestShape` Example: VB .NET

```
Module TestShape


    Sub Main()

        Dim myCircle As New   Circle(2.2)
        Dim myRectangle As New Rectangle(2.2, 3.3)
        Dim result As Double

        result = myCircle.calcArea()
        System.Console.Write("Circle area =  ")
        System.Console.WriteLine(result)

        result = myRectangle.calcArea()
        System.Console.Write("Rectangle area =  ")
```

```
        System.Console.WriteLine(result)

        System.Console.Read()
    End Sub

End Module
Public MustInherit Class Shape

    Protected area As Double

    Public MustOverride Function calcArea() As Double

End Class
Public Class Circle

    Inherits Shape

    Dim radius As Double

    Sub New(ByVal r As Double)

        radius = r

    End Sub

    Public Overrides Function calcArea() As Double

        area = 3.14 * (radius * radius)
        Return area

    End Function

End Class
Public Class Rectangle

    Inherits Shape

    Dim length As Double
    Dim width As Double

    Sub New(ByVal l As Double, ByVal w As Double)

        length = l
        width = w

    End Sub
```

```
    Public Overrides Function calcArea() As Double

        area = length * width
        Return area

    End Function

End Class
```

This page intentionally left blank

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 as well as 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 different 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. Only think about specific technologies 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. 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 OO code

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 that you learn a new language; however, making the move from COBOL to C++, C# .NET, Visual Basic .NET, or Java requires that you 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 go through the investment of learning OO concepts and the corresponding thought process first. 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 upon some of these concepts in Chapter 1, and here 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 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 simply 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 correctly and safely.

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

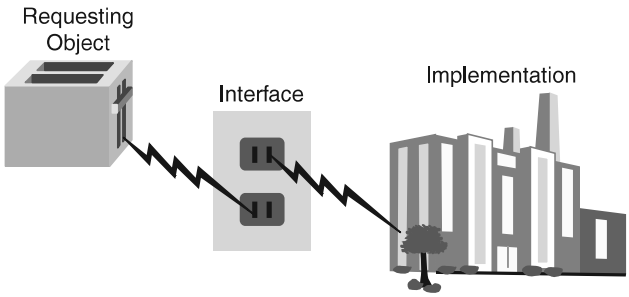


Figure 2.1 Power plant revisited.

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 for some eclectic people who love bucking the trends). On the other hand, as long as the performance and aesthetics didn't change, the average driver would not notice if 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 changing the interface.

What Users See

Interfaces also relate directly to classes. End users do not normally see any classes—they see the GUI or command line. However, programmers *would* see the class interfaces. Class reuse means that someone has already written a class. Thus, a programmer who uses a class must know how to get the class to work properly. This programmer will combine many classes to create a system. The programmer is the one who needs to understand the interfaces of a class. Therefore, 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 comprise 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 rule of thumb, the interface to a class should contain only what the user needs to know. In the toaster example, the user only needs to know 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. 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 dial a number. Yet, if the provider changes equipment, they don't change the way you make a call. The interface stays the same regardless of how the implementation changes. Actually, 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. There is one very important caveat to 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, there is a problem. 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 importantly, 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.

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.

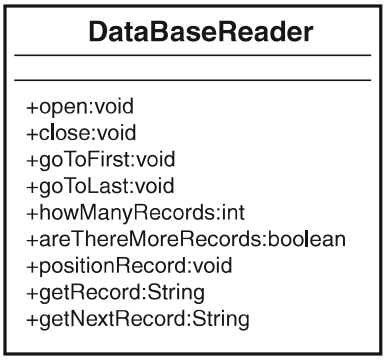


Figure 2.2 A Unified Modeling Language class diagram for the **`DataBaseReader`** class.

Public Interface

Remember that if a method is public, 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—this term is discussed later.

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 actually 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 there are any more records 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. There are OO databases on the market today that are perfect for OO applications. However, there is one small problem: 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 as long as relational databases are so prevalent. There might be an argument stating 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. This is due to the fact that 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. Of course, when this class is complete, it will probably contain more methods, and it will certainly contain attributes. However, as a programmer using this class, you do not need to know anything about these private methods and attributes. You certainly don't need to know what the code looks like within the public methods. You simply need to know how to interact with the interfaces.

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

```
public void open(String Name){

        /* Some application-specific processing */

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

        /* Some more application-specific processing */

};
```

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

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

Now the code looks like this:

```
public void open(String Name){

        /* Some application-specific processing

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

        /* Some more application-specific processing */

};
```

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

Code Recompilation

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

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

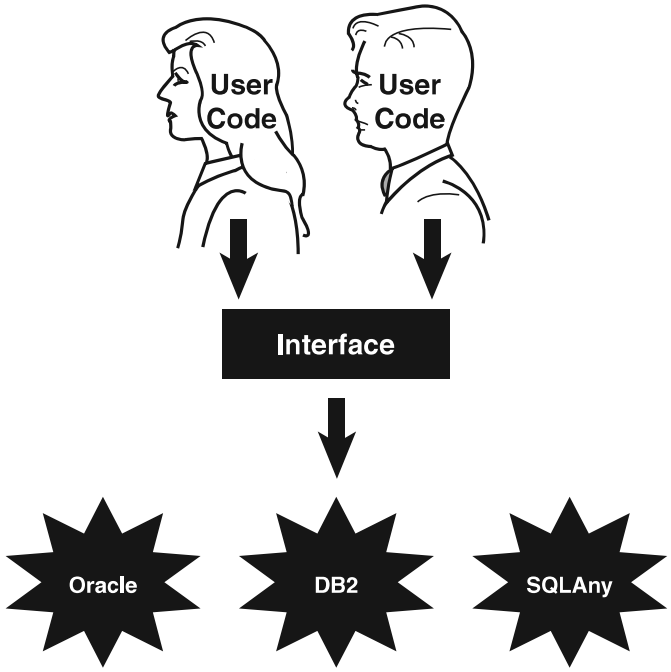


Figure 2.3 The interface.

Using Abstract Thinking When Designing Interfaces

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

It is possible to write a very useful, concrete class that is not at all reusable. This happens all the time, and there is nothing 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.

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 di-

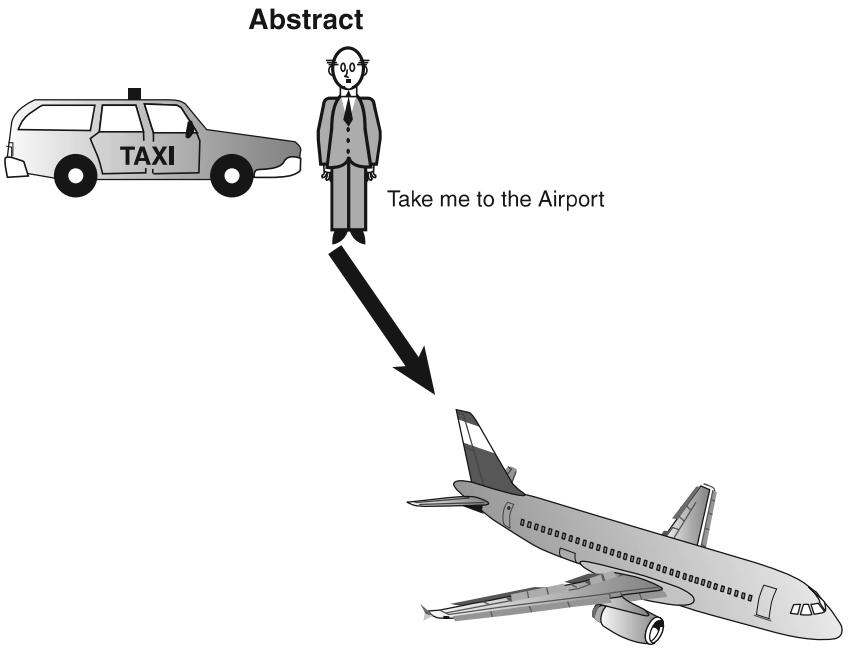


Figure 2.4 An abstract interface.

rection to turn, as illustrated in Figure 2.5. How the cabbie implements the actual drive is of no concern to you, the passenger. (Of course, the fare might become an issue at some point, if the cabbie cheats and takes you the long way to the airport.)

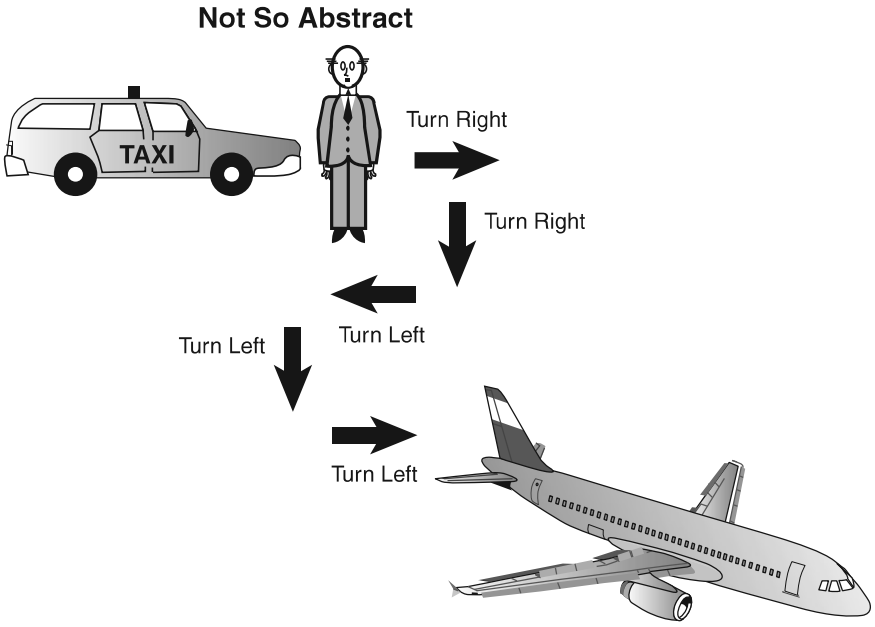


Figure 2.5 A not-so-abstract interface.

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 only work 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.

Giving the User the Minimal Interface Possible

When designing a class, the rule of thumb 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. There are times when it is problematic for the user to have 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. Of course, 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. 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, like "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.

For a software analogy, consider that users might want a programmer to provide a certain function. However, if the programmer finds the request technically impossible, the request can't be satisfied, no matter how much the programmer wants to help.

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.

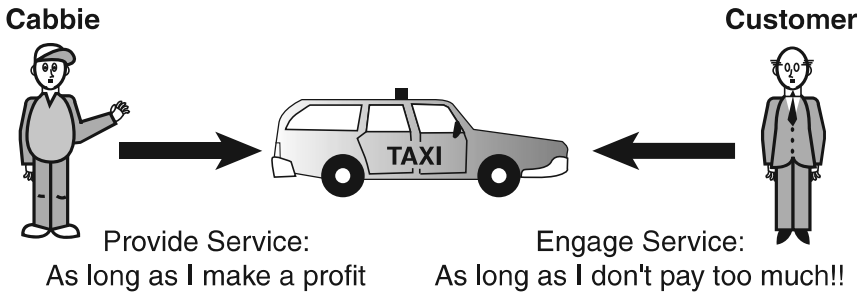


Figure 2.6 Providing services.

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 UseCases.

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.

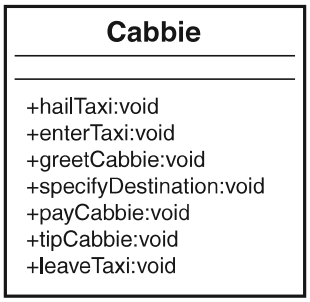


Figure 2.7 The methods in a cabbie class.

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, then 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 simply 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 actually 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 actually a part of the implementation because the user cannot see it. (The user should only see 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, "Advanced Object-Oriented Concepts," we' talk about the fact that the object has a life cycle: It is born, it lives, and it dies. While it is alive, it might transition through many different 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

Meyers, Scott. *Effective C++*, 3rd ed. Addison-Wesley Professional, 2005. Boston, MA.

Flower, Martin. *UML Distilled*, 3rd ed. Addison-Wesley Professional, 2003. Boston, MA.

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

This page intentionally left blank

3

Advanced Object-Oriented Concepts

Chapters 1, "An Introduction to Object-Oriented Concepts," and 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 several more advanced OO concepts such as constructors, operator overloading and multiple inheritance. We also will consider error-handling techniques, the importance of understudying 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 actually involved in the design and implementation of an OO system.

Constructors

Constructors are a new concept for people doing structured programming. Constructors do not normally exist in non-OO languages such as COBOL, C and Basic. In the first two chapters we alluded to special methods that are used to *construct* objects. In OO languages, constructors are methods that share the same name as the class and have no return type. For example, a constructor for the `Cabbie` class 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 again that 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.

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 de-compile 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.

> **Providing a Constructor**
>
> The rule of thumb is that you should *always* provide a constructor, even if you do not plan on doing 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, 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 maintenance is performed on the class that added another constructor, then the default constrictor is not created. In short, the default constructor is only added if you don't include one. As soon as you include just one, the default constructor is not included.

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 simply 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).

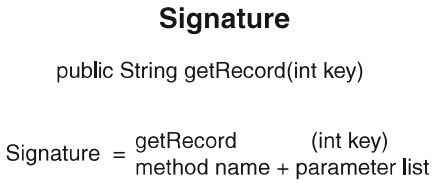


Figure 3.1 The components of a signature.

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.

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 listed below.

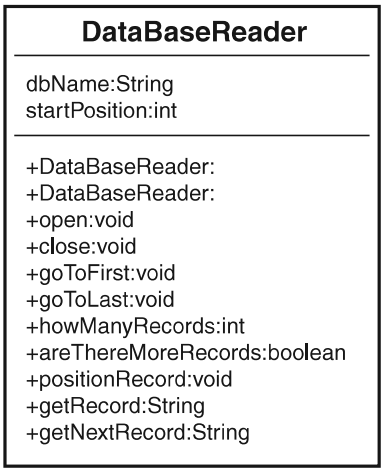


Figure 3.2 The **`DataBaseReader`** class diagram.

> **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):

```
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, like `0`.

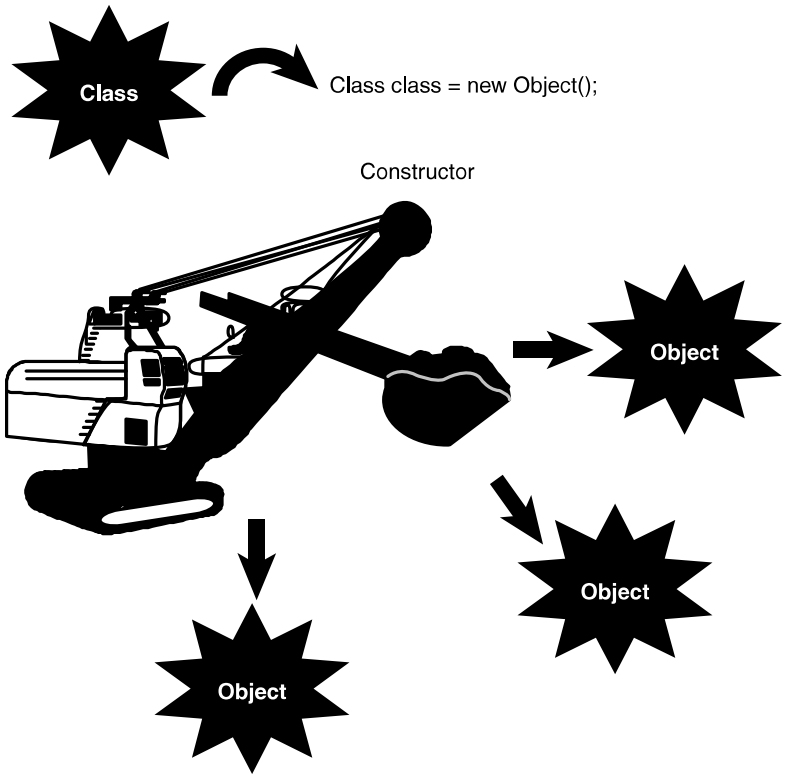


Figure 3.3 Creating a new object.

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):

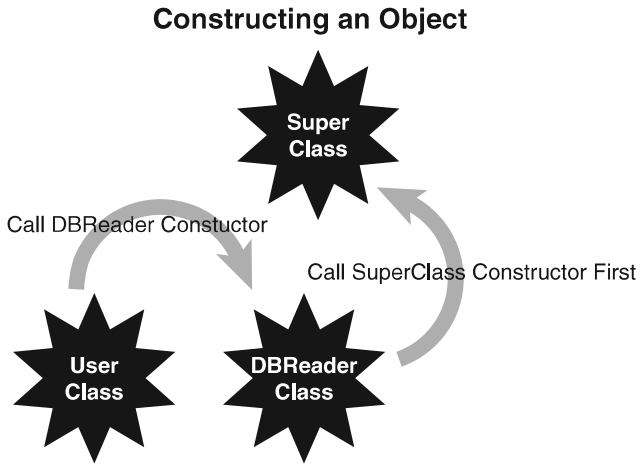


Figure 3.4 Constructing an object.

1. The first thing that happens inside the constructor is that 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. Then 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. Then the rest of the code in the constructor executes.

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 the fact 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 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 courting danger.

Assuming that your code has the ability to detect and trap an error condition, you can handle the error in several different ways: On page 223 of their book *Java Primer Plus*, Tyma, Torok, and Downing 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. On page 139 of their book *Object-Oriented Design in Java*, 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? The bottom line is that you should not ignore the 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. In the latter case, you might not even know you are getting incorrect results for some period of time.

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 there is a problem. 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.

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 `if` 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. You might need to prompt the user for 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 in the next section) can be expensive in terms of overhead. Thus, although exceptions are a great design choice, you will still want to consider other error handling techniques, depending on your design and performance needs.

Although this means of error checking is preferable to the previous solutions, it still has a few potentially limiting 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.

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# and C++, 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 `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 handle 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, i.e., 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 to say that exceptions are an important advantage for OO programming languages. Here is an example of how an exception is caught in Java:

```
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 just 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. However, because the exception was thrown within a `try` block, the `catch` block is checked to see whether the spe-

cific 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 (see Figure 3.5).

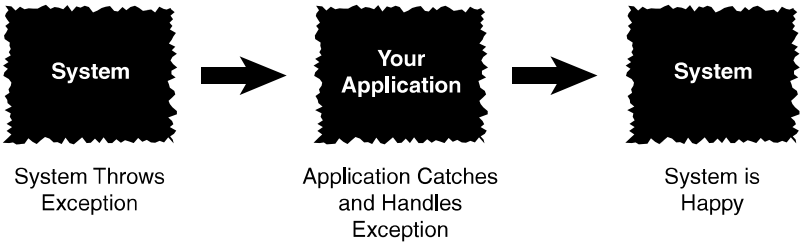


Figure 3.5 Catching an exception.

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 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 Concept 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 a class.

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#, and C++, 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;
    }

}
```

In this example, there are 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.

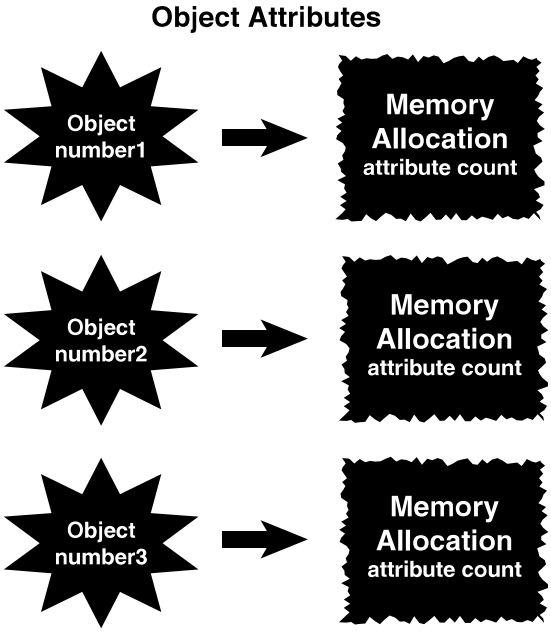


Figure 3.6 Object attributes.

Object Attributes

There are many design situations in which 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:

```
public class Number {

    int count;    // available to both method1 and method2

    public method1() {
```

```
        count = 1;
    }

    public method2() {
        count = 2;
    }

}
```

In this case, 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 actually 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:

```
public class Number {


    int count;

    public method1() {
        int count;
    }

    public method2() {
        int count;
    }

}
```

In this case, there are actually three totally separate memory locations with 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 there is some code that 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 to share attributes. In Java, C#, and C++, 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.

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 to keep 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.

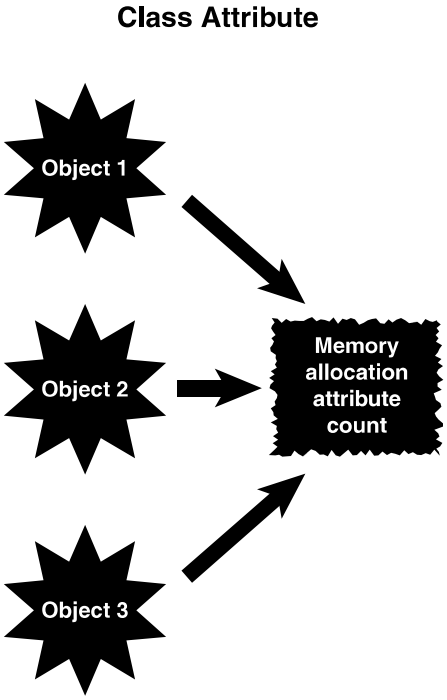


Figure 3.7 Class attributes.

Operator Overloading

Some OO languages allow you to overload an operator. C++ is an example of one such language. Operator overloading allows 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, there are times when 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 is 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;
```

More recent OO languages like Java and .NET do not allow operator overloading. While these languages do not allow the option of overloading operators; the languages themselves do overload the plus sign for string concatenation, but that's about it. The designers of Java must have decided that operator overloading was more of a problem than it was worth. If you must use operator overloading in C++, take care not to confuse the people who will use the class by documenting and commenting properly.

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 there are many real-world examples of multiple inheritance. 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.

As with operator overloading, the designers of Java and .NET decided that the increased complexity of allowing multiple inheritance far outweighed its advantages, so they eliminated it from the language. In some ways, the Java and .NET language construct of

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

Behavioral and Implementation Inheritance

Java and .NET interfaces are a mechanism for behavioral inheritance, whereas abstract classes are used for implementation inheritance. The bottom line is that Java and .NET interfaces provide 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. On page 34 of 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 or not. 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* is when all the references are followed and new copies are created for all referenced objects. There might be many levels 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 on page 265 of *Object-Oriented Design in Java* in a section called "Prefer a Tree to a Forest."

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

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

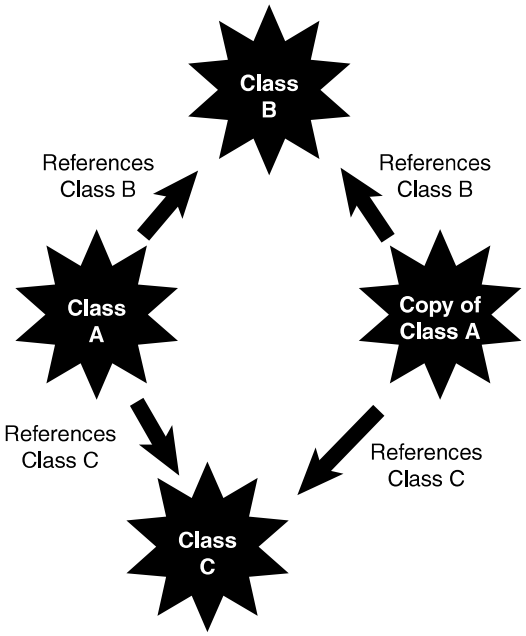


Figure 3.8 Following object references.

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

Meyers, Scott. *Effective C++*, 3rd ed. Addison-Wesley Professional, 2005. Boston, MA.

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

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

Example Code Used in This Chapter

The following code is presented in C# .NET and VB .NET. These examples correspond to the Java code that is listed inside the chapter itself.

The `TestNumber` Example: C# .NET

```
using System;

namespace TestNumber
{
    class Program
    {
        public static void Main()
        {

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

        }
    }

    public class Number
    {

        int count = 0;     // available to both method1 and method2

        public void method1()
        {
            count = 1;
        }

        public void method2()
        {
            count = 2;
        }

    }
}
```

The `TestNumber` Example: VB .NET

```
Module TestNumber

    Sub Main()

        Dim number1 As New Number()
        Dim number2 As New Number()
        Dim number3 As New Number()

        System.Console.ReadLine()

    End Sub

End Module
Public Class Number

    Dim count As Integer

    Function method1() As VariantType

        count = 1

    End Function

    Function method2() As VariantType

        count = 2

    End Function

End Class
```

This page intentionally left blank

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 an 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 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—in part to emphasize that the interface is the primary part 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 file name. If the names do not match, the application won't work.

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 {


}
```

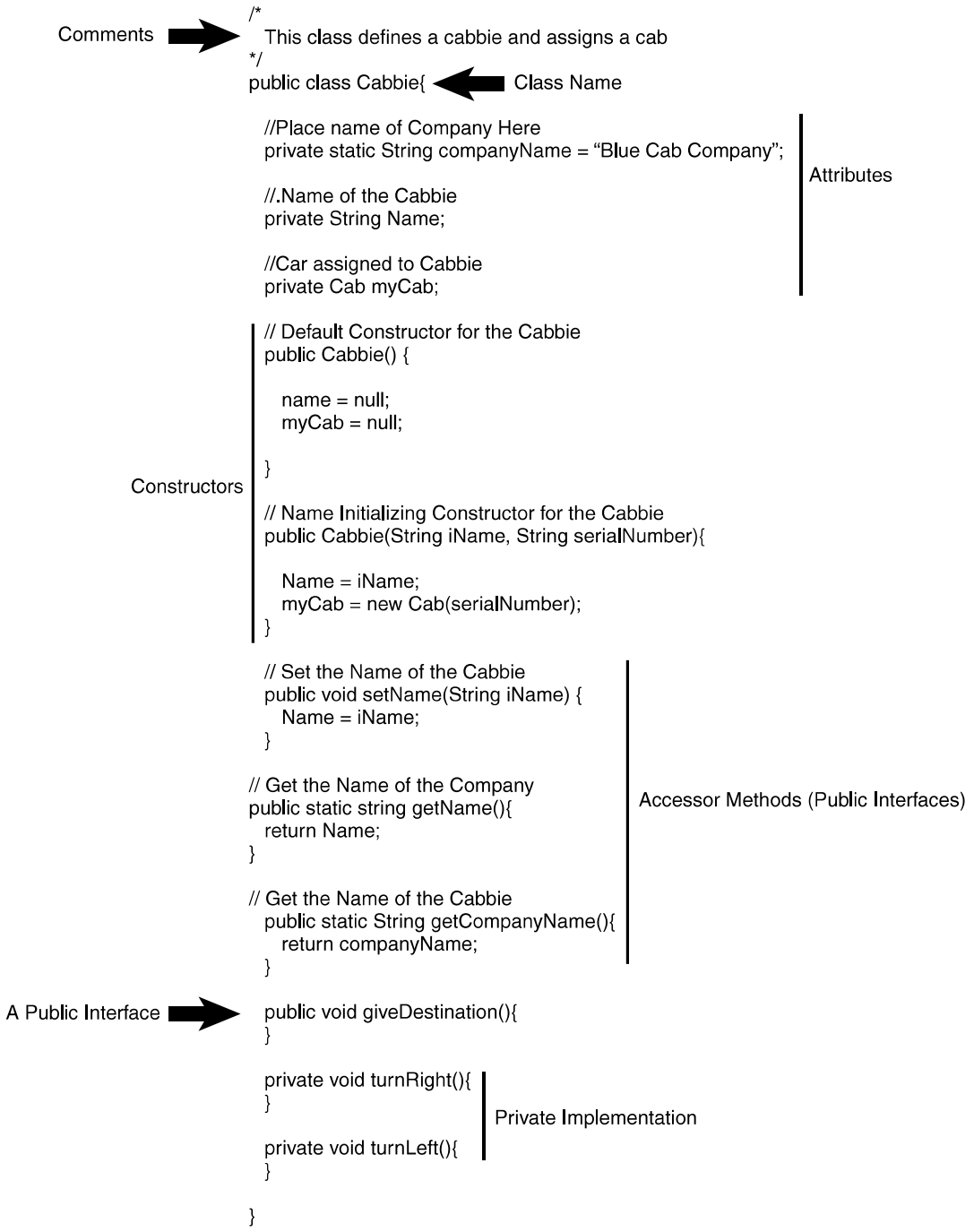


Figure 4.1 Our sample class.

Using Java Syntax

Remember that the convention for this book is to use Java syntax. The syntax will be similar but somewhat different in C#, .NET, VB .NET, or C++, and totally different in other OO languages such as Smalltalk.

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, C# .NET, and C++, there are two kinds of comments.

The Extra Java and C# Comment Style

In Java and C#, there are actually three types of comments. In Java, the third comment type (`/** */`) relates to a form of documentation that Java provides. We will not cover this type of comment in this book. C# provides similar syntax to create XML documents.

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:

```
/*

  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:

```
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, "Advanced Object-Oriented Concepts," for more discussion on class attributes.) Thus, even if 500 objects are instantiated from the `Cabbie` class, there will be only one copy in memory of the `companyName` attribute (see Figure 4.2).

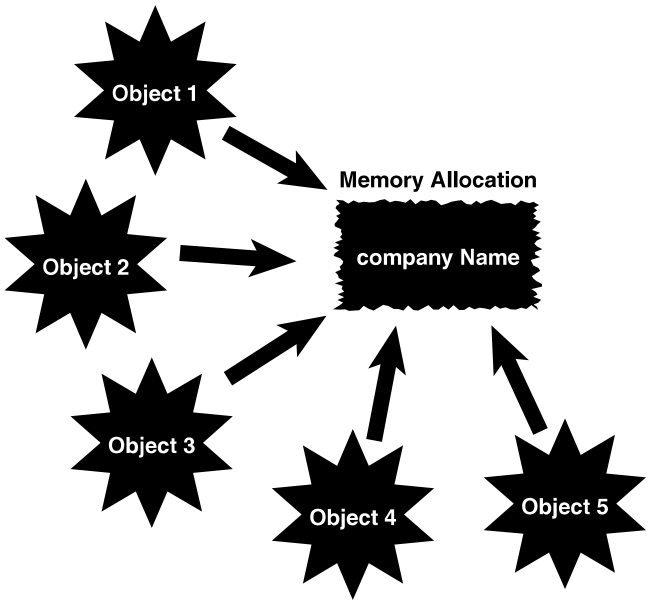


Figure 4.2 Object memory allocation.

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. Likewise, for the purposes of this example, 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. The compiler will provide a default constructor if you do not specify a constructor for this, or any, class. By definition, the reason it is called a default constructor here is because it is a constructor with no arguments. If you provide a constructor with arguments, the system will identify that you have provided a constructor and thus will not provide a default constructor. The rule is that the default constructor is only provided if you provide *no* constructors in your code.

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 technique. 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 actually 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.

As we know, it is always a good idea 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 un-initialized reference as if it were properly initialized.

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

```
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 actually 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—objects that want to instantiate a `Cab` object.

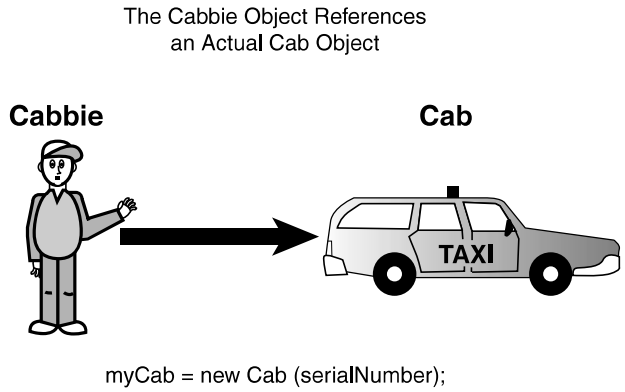


Figure 4.3 The **`Cabbie`** object referencing an actual cab object.

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 appropri-

ate information. Isn't it necessary to inspect and sometimes change another class's attribute? The answer is yes, of course. There are 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 really boil down to data integrity and efficient debugging.

Assume that there is a bug 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 only have to look in 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 simply 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 incorrect value. In general, the setters are used to ensure a level of data integrity.

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

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.

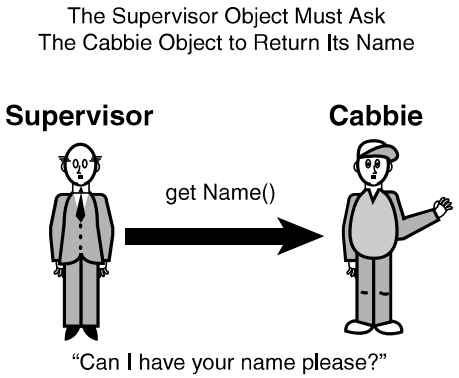


Figure 4.4 Asking for information.

Objects

Actually, there isn't a physical copy of each non-static 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.

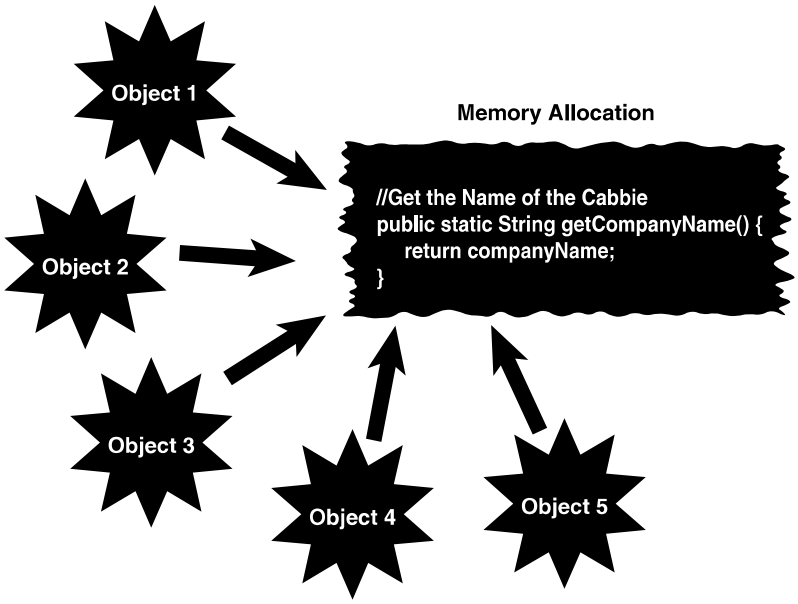


Figure 4.5 Method memory allocation.

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.

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

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 of this method is not important at this time. The main point here 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. `Some` methods in a class may be hidden from other classes. These methods are declared as `private`:

```
private void turnRight(){
}

private void turnLeft() {
}
```

These private methods are simply 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 only will use 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

Flower, Martin. *UML Distilled*, 3rd ed. Addison-Wesley Professional, 2003. Boston, MA.

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

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

Example Code Used in This Chapter

The following code is presented in C# .NET and VB .NET. These examples correspond to the Java code that is listed inside the chapter itself.

The `TestCab` Example: C# .NET

```
using System;
namespace ConsoleApplication1
{
    class TestPerson
    {
        public static void Main()
        {
            Cabbie joe = new Cabbie("Joe", "1234");

            Console.WriteLine(joe.Name);
            Console.ReadLine();
        }
    }

    public class Cabbie
    {
```

```
    private string strName;
    private Cab myCab;

    public Cabbie() {

         name = null;
              myCab = null;

      }

    public Cabbie(string iName, string serialNumber) {

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

    }

        //Methods
        public String Name
        {
            get { return strName; }
            set { strName = value; }
        }

    }

    public class Cab
    {

        private string serialNumber;

           public Cab (string sn) {

            serialNumber = sn;

        }

    }

}
```

The `TestCab` Example: VB .NET

```
Module TestCabbie

    Sub Main()
```

```
        Dim joe As New Cabbie("Joe", 1234)

        joe.Name = "joe"

        Console.WriteLine(joe.Name)

        Console.ReadLine()

    End Sub

End Module
Public Class Cabbie

    Dim strName As String

    Sub New()

        strName = " "

    End Sub

    Sub New(ByVal iName As String, ByVal serialNumber As String)

        strName = iName
        Dim myCab As New Cab(serialNumber)

    End Sub

    Public Property Name() As String
        Get
            Return strName
        End Get
        Set(ByVal value As String)
            strName = value
        End Set
    End Property

End Class
Public Class Cab

    Dim serialNumber As String

    Sub New(ByVal val As String)

        serialNumber = val

    End Sub

End Class
```

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 solid classes. Obviously, no list such as this can be considered complete. You will undoubtedly add many guidelines to your personal list as well as 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.

When moving to OO development for the first time, many people tend to still think in a structured way. One of the primary mistakes is to create a class that has behavior but no class data. In effect, they are creating a set of functions or subroutines in the structured model. This is not what you want to do because it doesn't take advantage of the power of encapsulation.

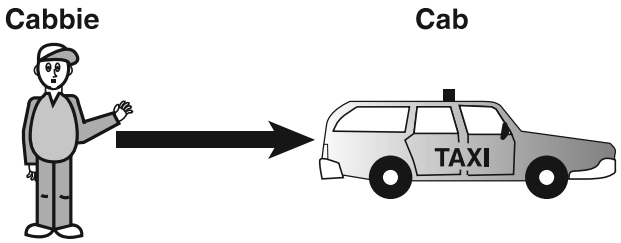


Figure 5.1 A cabbie and a cab are real-world objects.

> **Note**
>
> One of the better books pertaining to class design guidelines and suggestions is *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.

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. On page 109 of 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 by means of inheritance. In short, if designed with inheritance in mind, a new class can inherit from an existing class and create a new class with an extended interface.

To illustrate, consider the cabbie example once 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).

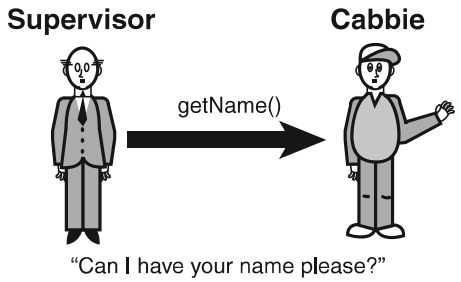


Figure 5.2 The public interface specifies how the objects interact.

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 them 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. Of course, 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. For example, a change to the implementation should not necessitate a change in user's application code.

In the cabbie example, the `Cabbie` class might contain behavior pertaining to how it eats breakfast. However, the cabbie's supervisor does not need to know what the cabbie has for breakfast. 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 is accessible from other objects.

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, "Advanced Object-Oriented Concepts." Revisit this discussion if you need a refresher on guidelines for designing constructors.

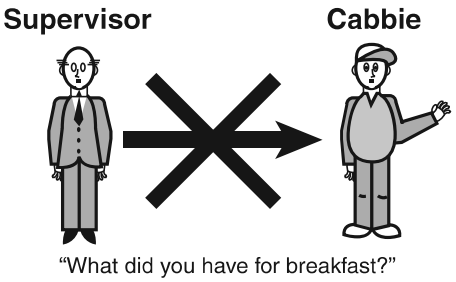


Figure 5.3 Objects don't need to know some implementation details.

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.

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 simply 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 rule of thumb is that the application should never crash. When an error is encountered, the system should either fix itself and continue, or 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 every book and article, 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. The bottom line is that 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 be lethal.

Many people have said all of 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 overcommenting can be a problem as well. Too much documentation and/or commenting can become noise and may defeat the purpose of the documentation in the first place. Unfocused documentation is, unfortunately, often ignored.

Building Objects with the Intent to Cooperate

In Chapter 6, "Designing with Objects," we discuss many of the issues involved in designing a system. We can safely say that almost no class lives in isolation. In most cases, there is no reason to build a class if it is not going to interact with other classes. This is simply 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).

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 figure out all the possible scenarios in which a `Cabbie` object must operate is not a trivial task.

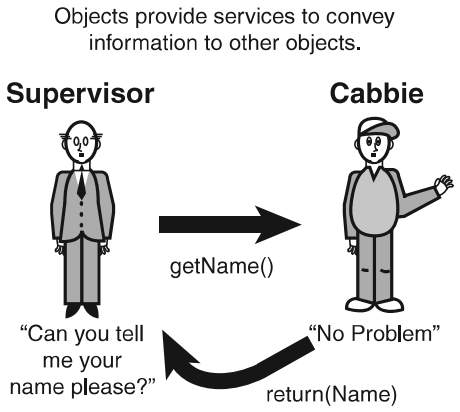


Figure 5.4 Objects should request information.

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 `Vendor` 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 `Vendor` (assuming, of course, 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.

What Attributes and Methods Can Be Static?

It is important to decide what attributes and methods can be declared as static. Revisit the discussions in Chapter 3 on using the `static` keyword to understand how to design these into your classes—these attributes and methods are shared by all objects of a class.

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 conventions that might make sense to them, but are totally incomprehensible to others. Take care when forcing other to conform to a convention. Make sure that the conventions are sensible and that everyone involved understands the intent behind them.

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 code (that is, the code will only run 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).

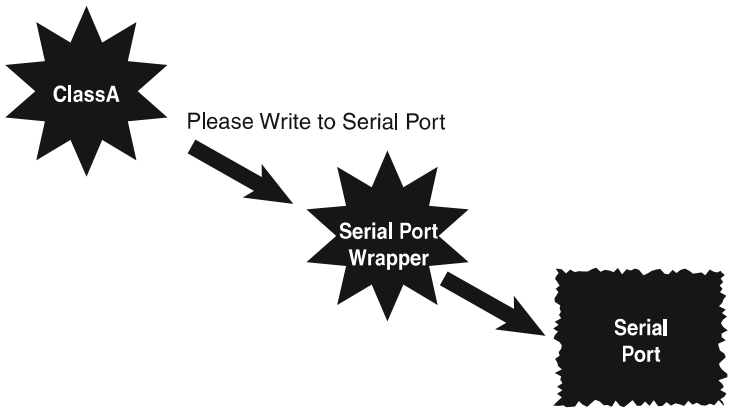


Figure 5.5 A serial port wrapper.

If the class moves to another hardware system, the way to access the serial port changes, or you want to go to a parallel port, the code in your primary class does not have to change. The only place the code needs to change is in the wrapper class.

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.

> **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 really 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.

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

```
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 only needed 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:

```
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.

A Class Should Be Responsible for Itself

In a training class based on their book, *Java Primer Plus*, Tyma, Torok, and Downing propose the class design guideline that all objects should be responsible for acting on themselves whenever possible. Consider trying to print a circle.

To illustrate, let's use a non-OO example. In this example, the print command is passed a `Circle` as an argument and prints it (see Figure 5.6):

```
print(circle);
```

The functions `print`, `draw`, and others need to have a `case` statement (or something like an `if/else` structure) to determine what to do for the given shape passed. In this case, a separate print routine for each shape could be called.

```
printCircle(circle);
printSquare(square);
```

Every time you add a new shape, all the functions need to add the shape to their `case` statements.

```
switch (shape) {
    case 1:  printCircle(circle); break;
    case 2:  printSquare(square); break;
    case 3:  printTriangle(triangle);  break;
    default: System.out.println("Invalid shape.");break;
}
```

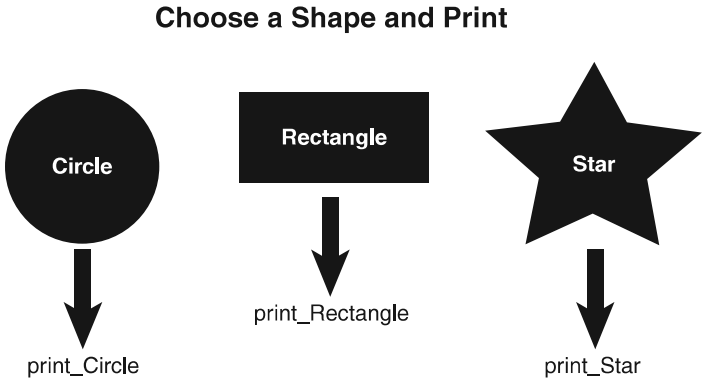


Figure 5.6 A non-OO example of a print scenario.

Now let's look at an OO example. By using polymorphism and grouping the `Circle` into a `Shape` category, `Shape` figures out that it is a `Circle` and knows how to print itself (see Figure 5.7):

```
Shape.print(); // Shape is actually a Circle
Shape.print(); // Shape is actually a Square
```

The important thing to understand here is that the call is identical; the context of the shape dictates how the system reacts.

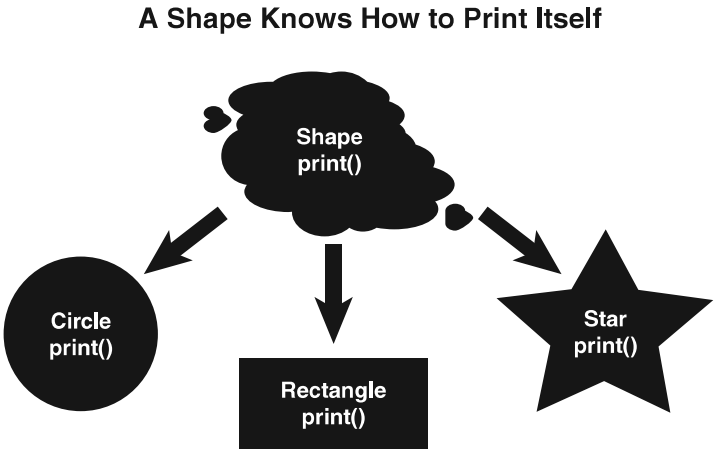


Figure 5.7 An OO example of a print scenario.

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 only be made 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. Simply finding all these method calls is a daunting task.

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

Using Iteration

As in most design and programming functions, using an iterative process is recommended. This dovetails well into the concept of providing minimal interfaces. 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 connect 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 really not necessary at this point. In fact, it might cost valuable time and energy to complete the implementation at this point because the design of the interface will affect the implementation, and the interface is not yet complete.

In Figure 5.8, 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.

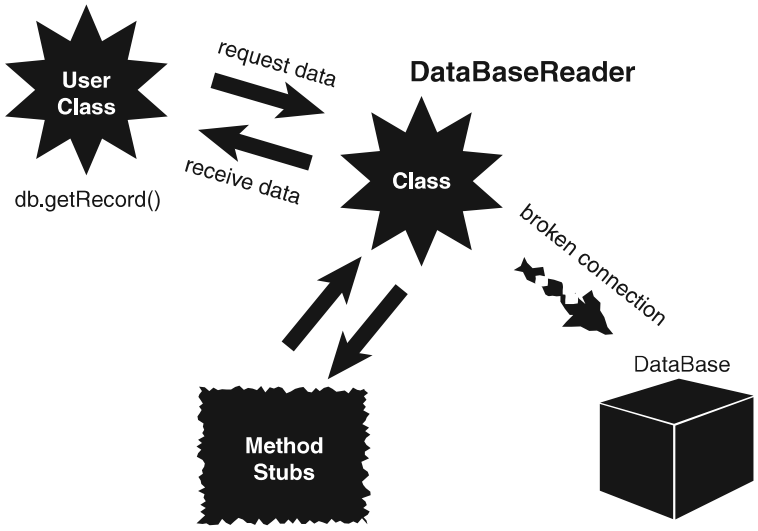


Figure 5.8 Using stubs.

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

```
public class DataBaseReader {

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

    private boolean DBOpen = 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 then 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. 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!

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 simply 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 considered 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 has very limited use.
- Relational database—Some sort of middleware is necessary to convert an object to a relational model.
- OO database—This is the logical way to make objects persistent, but most companies have all their data in legacy systems and are just starting to explore object databases. Even brand-new OO applications must usually interface with legacy data.

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 actually 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 de-serialization 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.

C# .Net and Visual Basic .NET provide the `ISerializable` interface, where the Microsoft documentation describes it as: Allows an object to control its own serialization and deserialization. All classes that are meant to be seriallized must implement this interface. The syntax for both C# .Net and Visual Basic .NET are listed in the following:

```
' Visual Basic .NET
Public Interface ISerializable

// C# .NET
public interface ISerializable
```

One of the problems with serialization is that it is often proprietary. The use of XML, which is discussed in detail later, is nonproprietary.

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

Meyers, Scott. *Effective C++*, 3rd ed. Addison-Wesley Professional, 2005. Boston, MA.

Ambler, Scott. *The Object Primer*, 3rd ed. Cambridge University Press, 2004. Cambridge, United Kingdom.

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

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

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

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

Example Code Used in This Chapter

The following code is presented in C# .NET and VB .NET. These examples correspond to the Java code that is listed inside the chapter itself.

The `TestMath` Example: C# .NET

```
public class Math {

    public int swap (int a, int b) {

        int temp=0;

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

        return temp;

    }

}
class TestMath {

    public static void Main() {
```

```
        Math myMath = new Math();
        MyMath.swap(2,3);
    }
}
```

The `TestMath` Example: VB .NET

```
Public Class Math

    Function swap(ByVal a As Integer, ByVal b As Integer)

        Dim temp As Integer

        temp = a
        a = b
        b = temp

        Return temp

    End Function

End Class

Module Module1

    Sub Main()

        Dim myMath As New Math()
     MyMath.sawp(2,3)

        System.Console.ReadLine()

    End Sub

End Module
```

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 tool. You have to 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.

Design Guidelines

One fallacy is that there is one true design methodology. This is not the case. There is no right or wrong way to create a design. There are many design methodologies available today, and they all have their proponents. However, the primary issue is not which design method to use, but simply whether to use a method at all. This can be expanded to the entire software development process. Many organizations do not follow a standard software development process. The most important factor in creating a good design is to find a process that you and your organization can feel comfortable with. 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 will generally include 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

In this chapter, we are most interested in the last item on this list. 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.

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.

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 there are other design models, such as rapid prototyping, that 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).

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

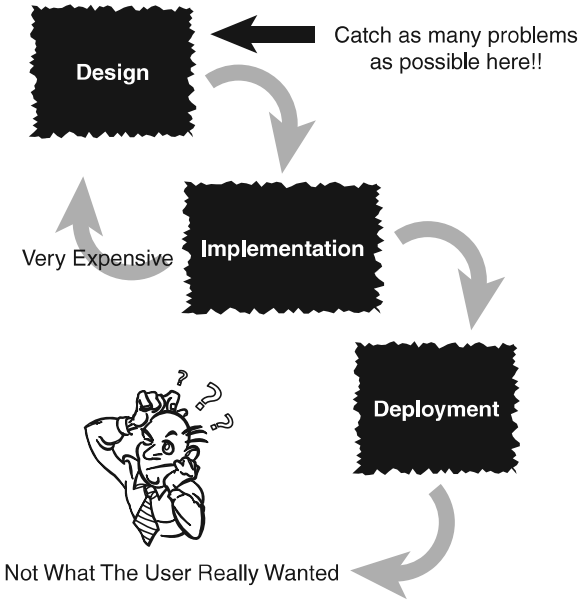


Figure 6.1 The waterfall method.

- 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 any case, it might be impossible to thoroughly test software, in the sense that absolutely *no* bugs exist. But that does not mean that we shouldn't try 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.

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 the 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 pre-release software simply because they are anxious to get the functionality the product promises.

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).

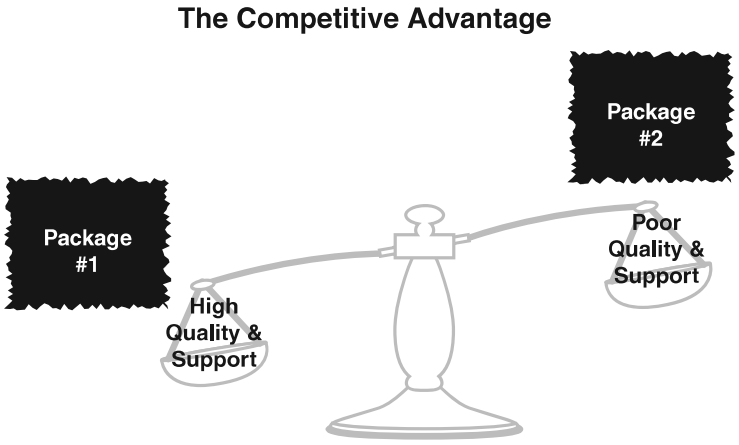


Figure 6.2 The competitive advantage.

Software Engineering

Although it might be acceptable to compare automobiles, bridges, and software when discussing quality, the legal implications of these topics cannot be compared, at least not yet. The legal issues regarding software are currently being defined and revised. Currently disclaimers such as "we are not responsible for anything that this software does or causes to happen" abound. Some other industries do not have this luxury. As the software legal process evolves and matures, software manufacturers may well have to contend with these issues. (As a standard disclaimer, in no way does this book attempt to offer any legal advice.)

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

There are a lot of variables 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 there is a valid reason 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 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 they want 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 Prototype of the User Interface

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, drawing the screens on a whiteboard or even on paper might be all that is needed. Traditionally, Visual Basic .NET is a good environment for prototyping, although other languages are now in play. 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 are the major concerns 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.

Case Study: A Blackjack Example

The rest of this chapter is dedicated to a case study pertaining to the design process covered in the previous sections. Walking through a case study seems to be a standard exercise in many object-oriented books that deal with OO design.

My first recollection of such an exercise was a graduate course that I took in which we followed an example in the book *Designing Object-Oriented Software* by Wrifs-Brock, Wilkerson, and Weiner. The modeling technique was called CRC modeling, which will be described later in this section. The case study was that of an automated teller machine (ATM) system. The iterative process of identifying the classes and responsibilities using CRC modeling was an eye-opening experience. The books, *The Object Primer* by Scott Ambler and *Object-Oriented Design in Java* by Gilbert and McCarty, both go through similar exercises using CRC modeling and Use Cases. Let's start an example that we expand on throughout this chapter.

Because we want to have some fun, let's create a program that simulates a game of blackjack. We will assume that the statement of work has already been completed. In fact, let's say that a customer has come to you with a proposal that includes a very well-written SOW and a rulebook about how to play blackjack.

According to the statement of work, the basic goal is to design a software system that will simulate the game of blackjack (see Figure 6.3). Remember, we will not describe how

to implement (code) this game—we are only going to design the system. Ultimately, this will culminate in the discovery of the classes, along with their responsibilities and collaborations. After some intense analysis, we have determined the requirements of the system. In this case, we will use a requirements summary statement; however, we could have presented the requirements as bullets. Because this is a small system, a requirements summary statement might make more sense. However, in most large systems, a database of the requirements (in bulleted list format) would be more appropriate. Here is the requirements summary statement:

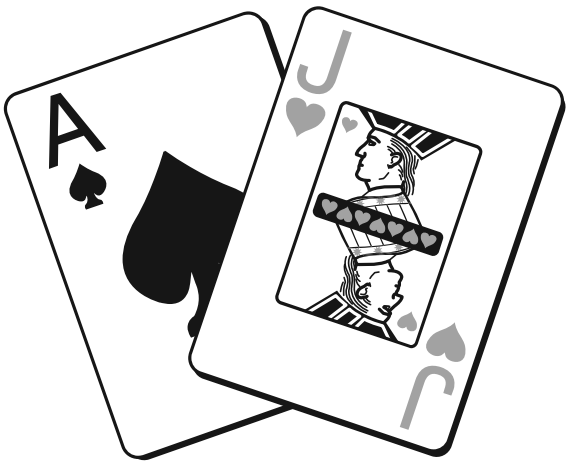


Figure 6.3 A winning blackjack hand.

Requirements Summary Statement
The intended purpose of this software application is to implement a game of blackjack. In the game of blackjack, one or more individuals play against the dealer (or house). Although there might be more than one player, each player plays only against the dealer, and not any of the other players.

From a player's perspective, the goal of the game is to draw cards from the deck until the sum of the face value of all the cards equals 21 or as close to 21 as possible, without exceeding 21. If the sum of the face value of all the cards exceeds 21, the player loses. If the sum of the face value of the first two cards equals 21, the player is said to have blackjack. The dealer plays the game along with the players. The dealer must deal the cards, present a player with additional cards, show all or part of a hand, calculate the value of all or part of a hand, calculate the number of cards in a hand, determine the winner, and start a new hand.

A card must know what its face value is and be able to report this value. The suit of the card is of no importance (but it might be for another game in the future). All cards must be members of a deck of cards. This deck must have the functionality to deal the next card, as well as report how many cards remain in the deck.

During the game, a player can request that a card be dealt to his or her hand. The player must be able to display the hand, calculate the face value of the hand, and determine the number of cards in the hand. When the dealer asks the player whether to deal another card or to start a new game, the player must respond.

Each card has its own face value (suit does not factor into the face value). Aces count as 1 or 11. Face cards (Jack, Queen, King) each count as 10. The rest of the cards represent their face values.

The rules of the game state that if the sum of the face value of the player's cards is closer to 21 than the sum of the face value of the dealer's cards, the player wins an amount equal to the bet that was made. If the player wins with a blackjack, the player wins 3:2 times the bet made (assuming that the dealer does not also have blackjack). If the sum of the face value of the player's cards exceeds 21, the bet is lost. Blackjack (an ace and a face card or a 10) beats other combinations of 21.

If the player and the dealer have identical scores and at least 17, it is considered a draw, and the player retains the bet.

As already mentioned, you could also have presented the requirements in bullet form, as we did for the `DataBaseReader` class in Chapter 2, "How to Think in Terms of Objects."

We want to take the perspective of the user. Because we are not interested in the implementation, we'll concentrate on the interface. Think back to the black-box example from Chapter 1, "Introduction to Object-Oriented Concepts." We only care about *what* the system does, not *how* it does it.

The next step is to study the requirements summary statement and start identifying the classes. Before we actually start this process, let's define how we are going to model and track the classes that we ultimately identify.

Using CRC Cards

Discovering classes is not a trivial process. In the blackjack example we are working on, there will be relatively few classes because this is intended as an example. However, in most business systems, there could be dozens of classes—perhaps 100 or more. There must be a way to keep track of the classes as well as their interactions. One of the most popular methods for identifying and categorizing classes is to use *class-responsibility-collaboration cards (CRC)*. Each CRC card represents a single class's data attributes, responsibilities, and collaborations.

For me, one of the more endearing qualities of CRC cards is that they can be non-electronic (although there are computer applications around that model CRC cards). In their basic sense, CRC cards are, quite literally, a collection of standard index cards.

You need to create three sections on each card:

- The name of the class
- The responsibilities of the class
- The collaborations of the class

The use of CRC cards conjures up scenes of dimly lit rooms—partially filled boxes of pizza, pop cans, and multitudes of index cards strewn around the room. Although this might be partially true, using CRC cards is a good technique because many of the people involved with the design will not be developers. They might not even have much com-

puter experience. Thus, using the index cards to discover classes (even a computerized CRC system) is a technique that everyone can understand. There are certainly various ways to perform these tasks, and many developers will use techniques that they are comfortable with. Figure 6.4 shows the format of a CRC card.

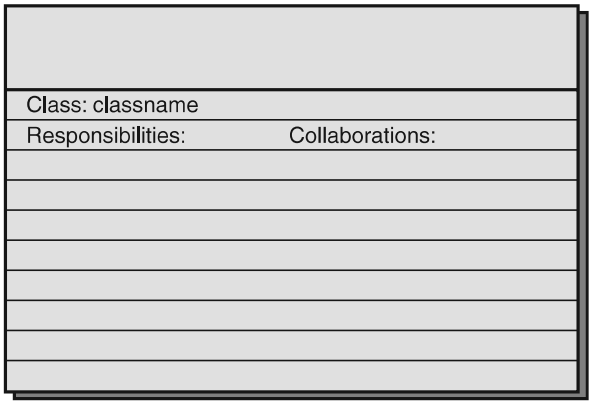


Figure 6.4 The format of a CRC card.

Identifying the Blackjack Classes

Remember that, in general, classes correspond to nouns, which are objects—people, places, and things. If you go through the requirements summary statement and highlight all the nouns, you have a good list from which you can start gleaning your objects.

Nouns

Although it is true that nouns generally indicate classes, nouns are not the only places where classes are found.

As stated earlier, you shouldn't get too hung up in getting things right the first time. Not all the classes that you identify from the list of nouns or elsewhere will make it through to the final cut. On the other hand, some classes that were not in your original list might actually make the final cut. Start feeling comfortable with the iterative process throughout the design. And as always, make sure that you realize that there are always many ways to solve a problem. It is often said that if you put 10 people in different rooms, they will come up with 10 different designs, and they might all be equally good. In most cases, although the designs might be different, ideally there will be significant overlap. Of course, when working with a team, the final design will have to be a consensus, iterating and evolving to a common solution.

Let's identify some nouns from our blackjack example: If the player and the dealer have identical scores and at least 17, it is considered a draw, and the player retains the bet.

Now let's make a list of the possible objects (classes):

- Game
- Blackjack
- Dealer
- House
- Players
- Player
- Cards
- Card
- Deck
- Hand
- Face value
- Suit
- Winner
- Ace
- Face card
- King
- Queen
- Jack
- Game
- Bet

Can you find any other possible classes that were missed? There might well be some classes that you feel should be in the list but are not. There might also be classes that you feel should not have made the list. In any event, we now have a starting point, and we can begin the process of fine-tuning the list of classes. This is an iterative process, and although we have 19 potential classes, our final class list might be a lot shorter.

Again, remember that this is just the initial pass. You will want to iterate through this a number of times and make changes. You might even find that you left an important object out or that you need to split one object into two objects. Now let's explore each of the possible classes (as classes are eliminated, they will be crossed out with a strikethrough):

- `Game`—*Blackjack* is the name of the game. Thus, we treat this in the same way we treated the noun *game*.
- ~~`Blackjack`~~—In this case, *game* might be considered a noun, but the game is actually the system itself, so we will eliminate this as a potential class.
- `Dealer`—Because we cannot do without a dealer, we will keep this one (as a note, we could abstract out the stuff pertaining to people in general, but we won't in this example). It might also be possible to avoid a dealer class altogether, thus having the dealer simply be an instance of the player class. However, there are enough additional attributes of a dealer that we should probably keep this class.
- ~~`House`~~—This one is easy because it is just another name for the dealer, so we strike it.
- ~~`Players`~~ and `player`—We need players, so we have to have this class. However, we want the class to represent a single player and not a group of players, so we strike ~~`player`~~ and keep `player`.

- ~~`Cards`~~ and `card`—This one follows the same logic as `player`. We absolutely need cards in the game, but we want the class to represent a single card, so we strike ~~`cards`~~ and keep `card`.
- `Deck`—Because there are a lot of actions required by a `deck` (like shuffling and drawing), we decide that this is a good choice for a class.
- `Hand`—This class represents a gray area. Each player will have a hand. In this game, we will require that a player has a single hand. So it would be possible for a player to keep track of the cards without having to have a `hand` object. However, because it is theoretically possible for a player to have multiple hands, and because we might want to use the concept of a hand in other card games, we will keep this class. Remember that one of the goals of a design is to be extensible. If we create a good design for the blackjack game, perhaps we can reuse the classes later for other card games.
- ~~`Face value`~~—The face value of the card is best represented as an attribute in the `card` class, so let's strike this as a class.
- ~~`Suit`~~—Again, this is a gray area. For the blackjack game, we do not need to keep track of the suit. However, there are card games that need to keep track of the suit. Thus, to make this class reusable, we should track it. However, the suit is not a good candidate for a class. It should be an attribute of a card, so we will strike it as a class.
- ~~`Ace`~~—This could better be represented as an attribute of the `card` class, so let's strike it as a class.
- ~~`Face Card`~~—This could better be represented as attribute of the `card` class, so let's strike it as a class.
- ~~`King`~~—This could better be represented as attribute of the `card` class, so let's strike it as a class.
- ~~`Queen`~~—This could better be represented as attribute of the `card` class, so let's strike it as a class.
- ~~`Bet`~~—This class presents a dilemma. Technically you could play blackjack without a bet; however, the requirements statement clearly includes a bet in the description. The bet could be considered an attribute of the player in this case, but there are many other games where a player does not need to have a bet. In short, a bet is not a logical attribute of a player. Also abstracting out the bet is a good idea because we might want to bet various things. You can bet money, chips, your watch, your horse, or even the title to your house. Even though there might be many valid arguments not to make the bet a class, in this case we will.

We are left with six classes, as shown in Figure 6.5.

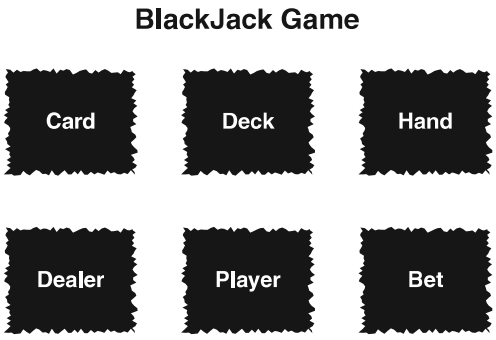


Figure 6.5 The initial blackjack classes.

Design Decisions

The dealer could be a specific type of player and perhaps inherit from a player class. However, this would be a design decision.

Identifying the Classes' Responsibilities

Responsibilities relate to actions. You can generally identify responsibilities by selecting the verbs from the summary of the requirements. From this list you can glean your responsibilities. However, keep in mind the following:

- Not all verbs in the requirements summary will ultimately end up as responsibilities.
- You might need to combine several verbs to find an actual responsibility.
- Some responsibilities ultimately chosen will not be in the original requirements summary.
- Because this is an iterative process, you need to keep revising and updating both the requirements summary and the responsibilities.
- If two or more classes share a responsibility, each class will have the responsibility.

Verbs

Although it is true that verbs generally correlate with responsibilities, verbs are not the only places where responsibilities are found.

Let's take an initial stab at identifying the verbs, which will lead us down the path toward uncovering the responsibilities of our classes: If the player and the dealer have identical scores and at least 17, then it is considered a draw, and the player retains the bet.

Now let's make a list of the possible responsibilities for our classes:

- `Card`
  - Know its face value
  - Know its suit
  - Know its value
  - Know whether it is a face card

    - Know whether it is an ace
    - Know whether it is a joker
- `Deck`
    - Shuffle
    - Deal the next card
    - Know how many cards are left in the deck
    - Know whether there is a full deck to begin
- `Hand`
    - Know how many cards are in the hand
    - Know the value of the hand
    - Show the hand
- `Dealer`
    - Deal the cards
    - Shuffle the deck
    - Give a card to a player
    - Show the dealer's hand
    - Calculate the value of the dealer's hand
    - Know the number of cards in the dealer's hand
    - Request a card (hit or hold)
    - Determine the winner
    - Start a new hand
- `Player`
    - Request a card (hit or hold)
    - Show the player's hand
    - Calculate the value of the player's hand
    - Know how many cards are in the hand
    - Know whether the hand value is over 21
    - Know whether the hand value is equal to 21 (and if it is a blackjack)
    - Know whether the hand value is below 21
- `Bet`
    - Know the type of bet
    - Know the value of the current bet
    - Know how much the player has left to bet
    - Know whether the bet can be covered

Remember that this is just the initial pass. You will want to iterate through this a number of times and make changes. You might even find that you've left an important responsibility out or that you need to split one responsibility into two. Now let's explore the possible responsibilities. We are left with the following classes and responsibilities:

`Card`

- Know its face value

  The card definitely needs to know this. Internally, this class must track the value of the card. Because this is an implementation issue, we don't want to phrase the responsibility in this way. From an interface perspective, let's call this *display face value*.

- Know its suit

  For the same reason as with face value, we will keep this responsibility, and rename it *display name* (which will identify the suit). However, we don't need this for blackjack. We will keep it for potential reuse purposes.

- ~~`Know whether it is a face card.`~~

  We could have a separate responsibility for face cards, aces, and jokers, but the report value responsibility can probably handle this. Strike this responsibility.

- ~~`Know whether it is an ace.`~~

  Same as previous item—let's strike this responsibility.

- ~~`Know whether it is a joker.`~~

  Same as previous, but notice that the joker was never mentioned in the requirements statement. This is a situation where we can add a responsibility to make the class more reusable. However, the responsibility for the joker goes to the report value, so let's strike this responsibility.

> **Class Design**
>
> What to do with the jokers presents an interesting OO design issue. Should there be two separate classes—a superclass representing a regular deck of cards (sans jokers) and a subclass representing a deck of cards with the addition of the jokers? From an OO purist's perspective, having two classes might be the right approach. However, having a single class with two separate constructors might also be a valid approach. What happens if you have decks of cards that use other configurations (such as no aces or no jacks)? Do we create a separate class for each, or do we handle them in the main class?
>
> This is another design issue that ultimately has no right or wrong answer.

`Deck`

- Shuffle

  We definitely need to shuffle the deck, so let's keep this one.

- Deal the next card

We definitely need to deal the next card, so let's keep this one.

- Know how many cards are left in the deck

  At least the dealer needs to know if there are any cards left, so let's keep this one.

- Know if there is a full deck to begin.

  The deck must know whether it includes all the cards. However, this might be strictly an internal implementation issue; in any event, let's keep this one for now.

`Hand`

- Know how many cards are in the hand

  We definitely need to know how many cards are in a hand, so let's keep this one. However, from an interface perspective, let's rename this *report the number of cards in the hand*.

- Know the value of the hand

  We definitely need to know the value of the hand, so let's keep this one. However, from an interface perspective, let's rename this *report the value of the hand*.

- Show the hand

  We need to be able to see the contents of the hand.

`Dealer`

- Deal the cards

  The dealer must be able to deal the initial hand, so let's keep this one.

- Shuffle the deck

  The dealer must be able to shuffle the deck, so let's keep this one. Actually, should we make the dealer request that the deck shuffle itself? Possibly.

- Give a card to a player

  The dealer must be able to add a card to a player's hand, so let's keep this one.

- Show the dealer's hand

  We definitely need this functionality, but this is a general function for all players, so perhaps the hand should show itself and the dealer should request this. Let's keep it for now.

- Calculate the value of the dealer's hand

  Same as previous. But the term *calculate* is an implementation issue in this case. Let's rename it *show the value of the dealer's hand*.

- Know the number of cards in the dealer's hand

  Is this the same as *show the value of the dealer's hand*? Let's keep this for now, but rename it *show the number of cards in the dealers hand*.

- Request a card (hit or hold)

A dealer must be able to request a card. However, because the dealer is also a player, is there a way that we can share the code? Although this is possible, for now we are going to treat a dealer and a player separately. Perhaps in another iteration through the design, we can use inheritance and factor out the commonality.

- Determine the winner

  This depends on whether we want the dealer to calculate this or the game object. For now, let's keep it.

- Start a new hand

  Let's keep this one for the same reason as the previous item.

`Player`

- Request a card (hit or hold)

  A player must be able to request a card, so let's keep this one.

- Show the player's hand

  We definitely need this functionality, but this is a general function for all players, so perhaps the hand should show itself and the dealer should request this. Let's keep this one for now.

- Calculate the value of the player's hand

  Same as previous. But the term *calculate* is an implementation issue in this case. Let's rename this *show the value of the player's hand*.

- Know how many cards are in the hand

  Is this the same as *show the player's hand*? Let's keep this for now, but rename it *show the number of cards in the player's hand*.

- Know whether the hand value is over 21, equal to 21 (including a blackjack), or below 21.

  Who should make this determination? These are based on the specific rules of the game. The player definitely needs to know this to make a decision about whether to request a card. In fact, the dealer needs to do this, too. This could be handled in *report the value of the hand*.

`Bet`

- Know the type of bet

  At this point, we will keep this for future reuse; however, for this game, we will require that the type of the bet is always money.

- Know the value of the current bet

  We need this to keep track of the value of the current bet. The player and the dealer need to know this. We will assume that the dealer (that is, the house) has an unlimited amount to bet.

- Know how much the player has left to bet

  In this case, the bet can also act as the pool of money that the player has available. In this way, the player cannot make a bet that exceeds his resources.

- Know whether the bet can be covered

  This is a simple response that allows the dealer (or the house) to determine whether the player can cover the bet.

As we iterate through the design process, we decide that we really do not want to have a separate bet class. If we need to, we can add it later. The decision needs to be based on two issues:

- Do we really need the class now or for future classes?
- Will it be easy to add later without a major redesign of the system?

After careful consideration, we decide that the bet class is not needed and most probably will not be needed later. We make an assumption that the payment method for all future bets will be money. An argument can be made that this approach might not be the best design decision. I can think of many reasons that we might want to have a bet object. There might be some behavior that should be encapsulated in a bet object. However, for now, we will scrap the bet object and make the dealer and players handle their own bets.

UML Use-Cases: Identifying the Collaborations

To identify the collaborations, we need to study the responsibilities and determine what other classes the object interacts with. In short, what other classes does this object need to fulfill all its required responsibilities and complete its job? As you examine the collaborations, you might find that you have missed some necessary classes or that some classes you initially identified are not needed:

- To help discover collaborations, use-case scenarios can be used. A *use-case* is a transaction or sequence of related operations that the system performs in response to a user request or event.
- For each use-case, identify the objects and the messages that it exchanges.

You might want to create collaboration diagrams to document this step. Obviously, there can be an infinite number of scenarios. The purpose of this part of the process is not necessarily to document all possible scenarios, which is an impossible task. The real purpose of creating use-case scenarios is to help you refine the choice of your classes and their responsibilities.

By examining the collaborations, you might identify an important class that you missed. If this is the case, you can simply add another CRC card. You might also discover that one of the classes you originally chose is not as important as you once thought, so you can strike it and remove the CRC card from consideration. CRC cards help you discover classes, whereas use-case scenarios help you discover collaborations.

For example, let's consider a single possible scenario. In this case, we have a dealer and a single player.

- Dealer shuffles deck
- Player makes bet
- Dealer deals initial cards
- Player adds cards to player's hand
- Dealer adds cards to dealer's hand
- Hand returns value of player's hand to player
- Hand returns value of dealer's hand to dealer
- Dealer asks player whether player wants another card
- Dealer deals player another card
- Player adds the card to player's hand
- Hand returns value of player's hand to player
- Dealer asks player whether player wants another card
- Dealer gets the value of the player's hand
- Dealer sends or requests bet value from players
- Player adds to/subtracts from player's bet attribute

Let's determine some of the collaborations. Assume that we have a main application that contains all the objects (that is, we do not have a `Game` class). As part of our design, we have the dealer start the game. Figures 6.6 through 6.15 present some collaboration diagrams pertaining to this initial design.

Figure 6.6 Start the game.

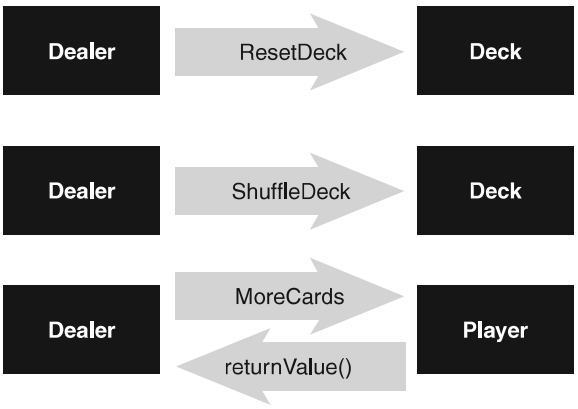


Figure 6.7 Shuffle and initially deal.

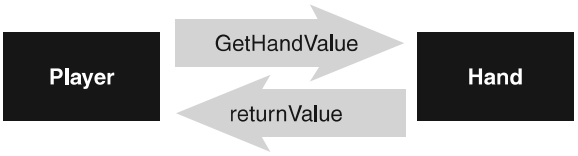


Figure 6.8 Get the hand value.

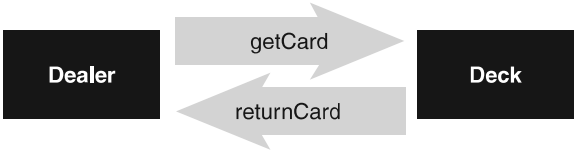


Figure 6.9 Get a card.

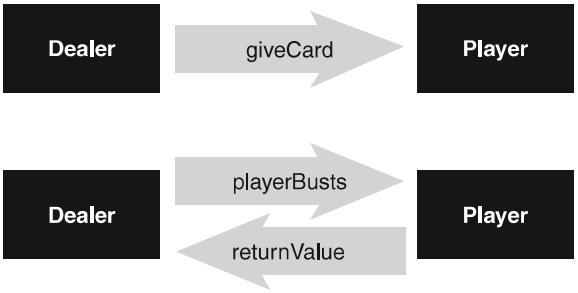


Figure 6.10 Deal a card and check to see whether the player busts.

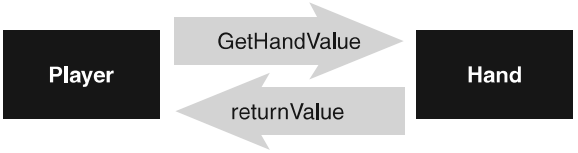


Figure 6.11 Return the value of the hand.

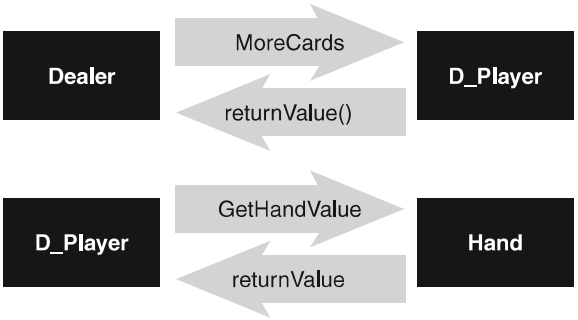


Figure 6.12 Does the dealer want more cards?

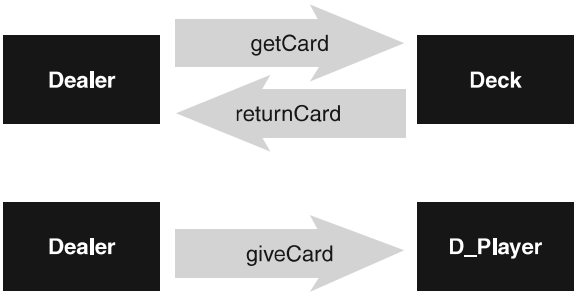


Figure 6.13 If requested, give the dealer a card.

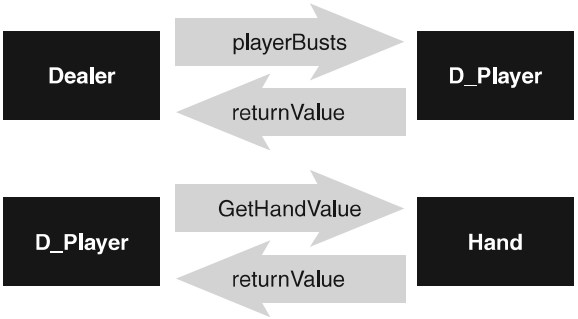


Figure 6.14 Does the dealer bust?

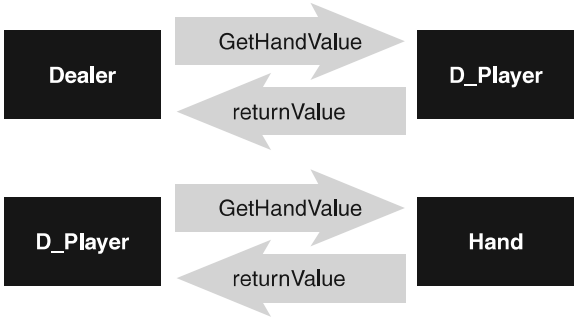


Figure 6.15 Do either the dealer or the player stand?

Intangibles

Be aware that there are more issues than the value of a player's hand involved in deciding whether to take another card. A player at a real blackjack table might go with a gut feel or how the dealer's hand looks. Although we might not be able to take gut feelings into consideration, we can deal with the issue of what the dealer's hand currently shows.

First Pass at CRC Cards

Now that we have identified the initial classes and the initial collaborations, we can complete the CRC cards for each class. It is important to note that these cards represent the initial pass only. In fact, although it is likely that many of the classes will survive the subsequent passes, the final list of classes and their corresponding collaborations might look nothing like what was gleaned from the initial pass. This exercise is meant to explain the process and create an initial pass, not to come up with a final design. Completing the design is a good exercise for you to undertake at the end of this chapter. Figures 6.16 through 6.20 present some CRC cards pertaining to this initial design.

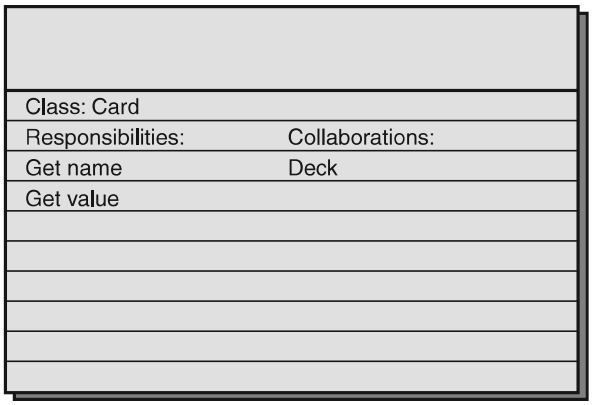


Figure 6.16 A CRC card for the **`Card`** class.

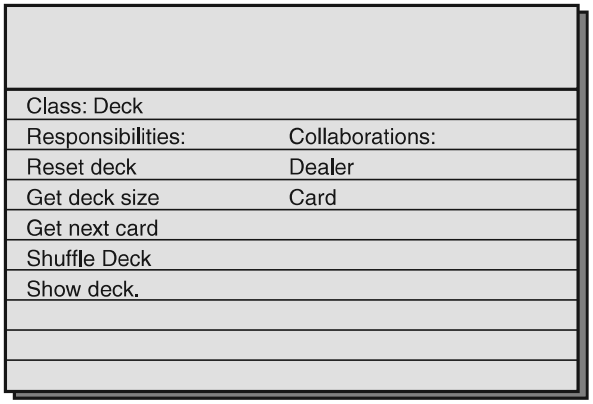
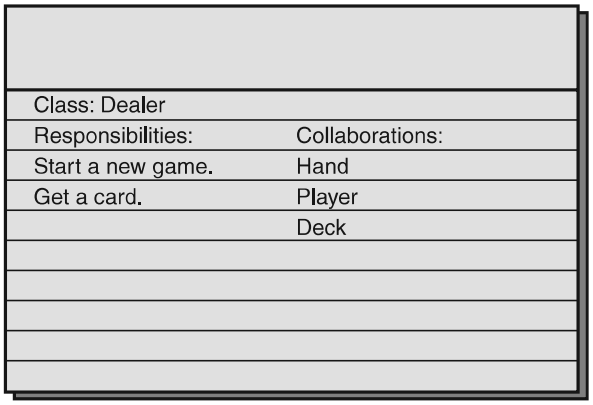
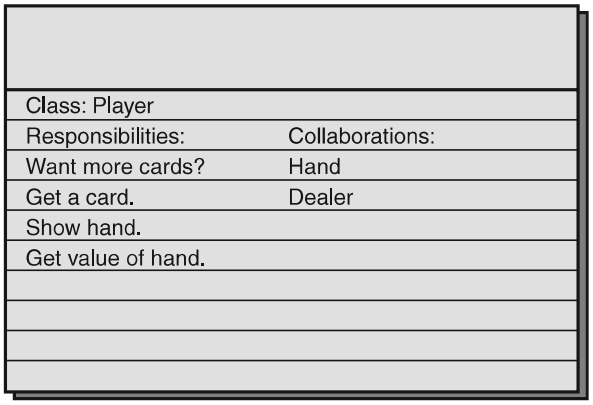


Figure 6.17 A CRC card for the **`Deck`** class.



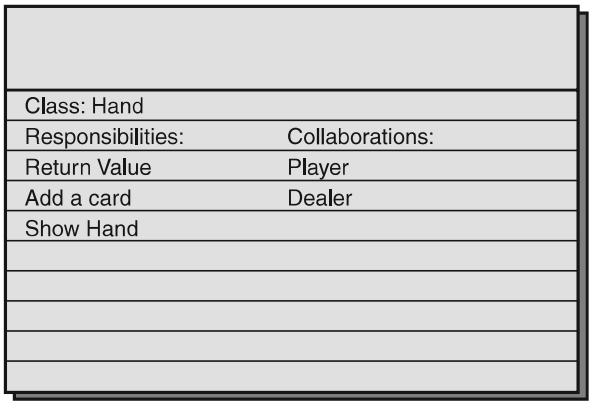
Class: Dealer	
Responsibilities:	Collaborations:
Start a new game.	Hand
Get a card.	Player
	Deck

Figure 6.18 A CRC card for the **`Dealer`** class.



Class: Player	
Responsibilities:	Collaborations:
Want more cards?	Hand
Get a card.	Dealer
Show hand.	
Get value of hand.	

Figure 6.19 A CRC card for the **`Player`** class.



Class: Hand	
Responsibilities:	Collaborations:
Return Value	Player
Add a card	Dealer
Show Hand	

Figure 6.20 A CRC card for the **`Hand`** class.

UML Class Diagrams: The Object Model

After you have completed the initial design using CRC cards, transfer the information contained on the CRC cards to class diagrams (see Figure 6.21). Note that this class diagram represents one possible design—it does not represent the initial pass of classes created during the previous exercise. The class diagrams go beyond the information on the CRC cards and might include such information as method parameters and return types. (Note that the UML diagrams in this book do not include method parameters.) Check out the options for the modeling tool that you have to see how information is presented. You can use the detailed form of the class diagram to document the implementation.

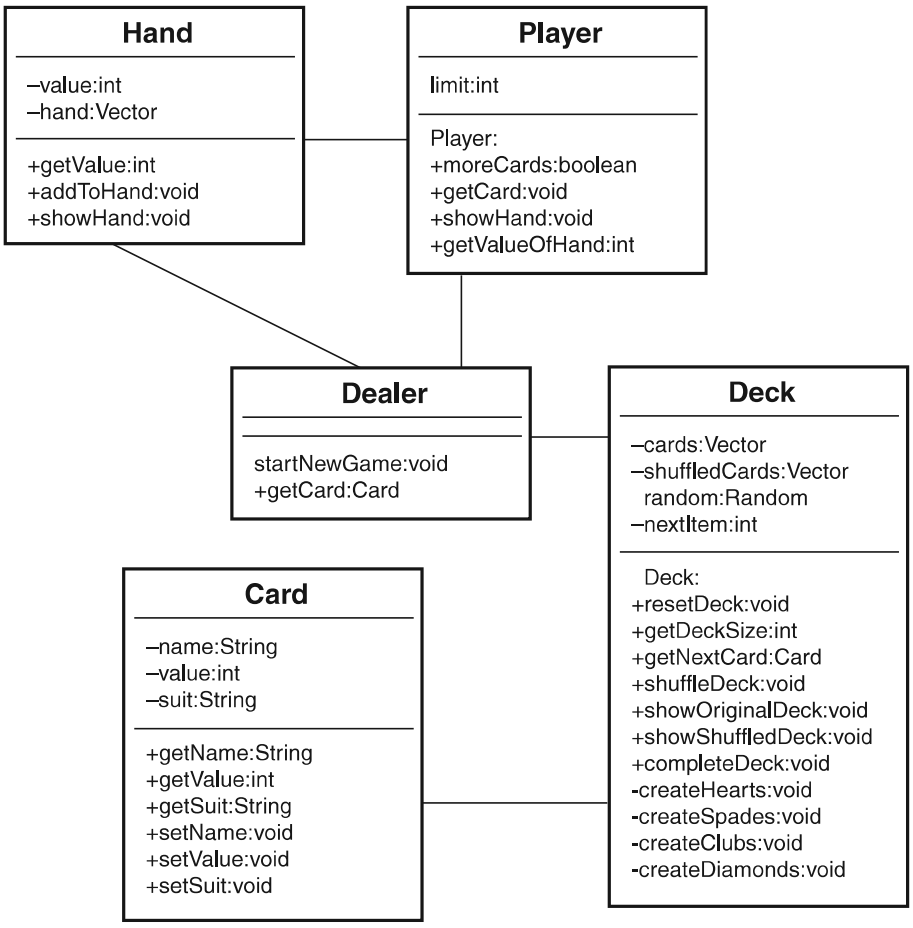


Figure 6.21 A UML diagram for the blackjack program.

Remember that the purpose of this exercise is to identify the classes and their interfaces. All the methods listed are public. Now a light bulb should be going off in your head.

Even though the search for the interfaces does not lead directly to private attributes and even private methods, the process is helpful in determining these as well. As you iter-

ate through the CRC process, note what attributes and private methods each class will require.

Prototyping the User Interface

As our final step in the OO 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 simply drawing it on paper or a whiteboard. You 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 you develop the user interface prototype, make sure that the users have the final say on the look and feel.

Conclusion

This chapter covers the design process for complete systems. It focuses on combining several classes to build a system. UML class diagrams represent this system. The example in this chapter shows the first pass at creating a design and is not meant to be a finished design. Much iteration may be required to get the system model to the point where you are comfortable with it.

> **Implementing Some Blackjack Code**
>
> While working on the first edition of this book, I came across a complete implementation of a blackjack game in the book Java 1.1 Developers Guide by Jamie Jaworski. If you would like to actually get your hands dirty and write some code to implement another blackjack design, you might want to pick up this book. It is a very good, compressive Java book.

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.