Beginning Object-Oriented Programming with VB 2005

From Novice to Professional

Daniel R. Clark

Apress®

Beginning Object-Oriented Programming with VB 2005: From Novice to Professional

Copyright © 2006 by Daniel R. Clark

ISBN (pbk): 1-59059-576-9

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jonathan Hassell
Technical Reviewer: Jon Box
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Manager: Beth Christmas
Copy Edit Manager: Nicole LeClerc
Copy Editors: Marilyn Smith and Ami Knox
Assistant Production Director: Kari Brooks-Copony
Production Editor: Janet Vail
Compositor: Kinetic Publishing Services, LLC
Proofreader: Christy Wagner
Indexer: Rebecca Plunkett
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

Contents at a Glance

Contents

PART 2 Object-Oriented Programming with Visual Basic

About the Author

DAN CLARK is a senior IT consultant specializing in .NET and SQL Server technologies. He is a Microsoft Certified Solution Developer and Microsoft Certified Database Administrator. For the past decade, he has been developing applications and training others how to develop applications using Microsoft technologies. Dan is a regular speaker at various developer conferences and user group meetings. He finds particular satisfaction in turning new developers on to the thrill of developing and designing object-oriented applications.

About the Technical Reviewer

As a Microsoft Regional Director and Chief Architect at ProTech Systems Group in Memphis, **JON BOX** serves as a .NET evangelist focused on delivering solutions and assisting developers to utilize .NET. Jon's current work emphasizes developing mobility solutions with Microsoft technologies, empowering software development with Visual Studio Team System, and building web sites with ASP.NET 2.0 and DotNetNuke. Being a presenter and a Microsoft MVP, Jon has given presentations at TechEd and MEDC, hosted MSDN web casts, spoken at various .NET user groups around the country, and serves on the INETA Speakers Bureau. Jon writes for the *.NET Developer's Journal*, coauthored *Building Solutions with the Microsoft .NET Compact Framework* with Dan Fox, and has a variety of mobility whitepapers on MSDN. Jon also is the cofounder and coleader of the Memphis .NET Users Group (`www.memphisdot.net`). You can see his current musings on technology at `http://jonbox.dotnetdevelopersjournal.com`.

Introduction

It has been my experience as a Visual Basic trainer that most people do not have trouble picking up the syntax of the language. What perplexes and frustrates many people are the higher-level concepts of object-oriented programming methodology and design. To compound the problem, most introductory programming books and training classes skim over these concepts or, worse, do not cover them at all. It is my hope that this book fills this void. My goal in writing this book is twofold. First, to provide you with the information needed to understand the fundamentals of programming with Visual Basic. Second and more importantly, to present you with the information required to master the higher-level concepts of object-oriented programming methodology and design.

This book provides you with the information needed to understand how you go about architecting an object-oriented programming solution aimed at solving a business problem. As you work your way through the book, first you will learn how to analyze the business requirements. Next, you will model the objects and relationships involved in the solution design. Finally, you will implement the solution using Visual Basic .NET. Along the way, you will learn the fundamentals of software design, the Unified Modeling Language (UML), object-oriented programming, Visual Basic (VB), and the .NET Framework.

Because this is an introductory book, it is meant to be a starting point for your study of the topics presented. As such, this book is *not* designed to make you an expert in object-oriented programming and UML; nor be an exhaustive discussion of VB and the .NET Framework; nor be an in-depth study of Visual Studio. It takes considerable time and effort to become proficient in any one of these areas. It is my hope that by reading this book, your first experiences in object-oriented programming will be enjoyable, comprehensible, and instill a desire for further study.

Target Audience

The target audience for this book is the beginning VB programmer who wants to gain a foundation in object-oriented programming along with the VB language basics. Programmers transitioning from a procedural-oriented programming model to an object-oriented model will also benefit from this book. In addition, there are many pre-.NET VB programmers who do not have a firm grasp of object-oriented programming. Now is the time to become acquainted with the fundamentals of object-oriented programming before transitioning to the current version of VB and the .NET Framework. Because the experience level of a “beginner” can vary immensely, I have included a primer in Appendix A, which discusses some basic programming tenets. I would suggest you review these concepts if you are new to programming.

Organization of the Book

This book is organized into three parts:

Part 1 delves into object-oriented programming methodology and design—concepts that transcend a particular programming language. The concepts presented are important to the success of an object-oriented programming solution regardless of the implementation language chosen. At the conclusion of this part, a case study walks you through modeling a "real-world" application.

Part 2 looks at how object-oriented programming is implemented in Visual Basic. You will look at creating class structures, creating hierarchies, and implementing interfaces. This part also introduces object interaction and collaboration. You will see how the object-oriented programming topics discussed in Part 1 are transformed into Visual Basic coding constructs.

Part 3 returns to the case study introduced and modeled at the end of Part 1. Using the knowledge gained in Part 2, you will transform the design into a fully functional VB application. This includes designing a graphical user interface, implementing the business logic, and integrating with a relational database to store data. Along the way you will be exposed to the .NET Framework classes used to work with data, and see how to create a Windows-based user interface, a Web-based user interface, and a Web service-based programmatic interface.

Activities and Software Requirements

One of the most important aspects of learning is doing. You cannot learn to ride a bike without jumping on a bike, and you cannot learn to program without "cranking out" code. Any successful training program needs to include both a theory component and a hands-on component. I have included both components throughout this book. It is my hope that you will take these activities seriously and work through them thoroughly and even repeatedly. Contrary to some students' perception that these activities are "exercises in typing," this is where the theory becomes concrete and true simulation of the concepts occurs. I also encourage you to play during the activities. Do not be afraid to alter some of the code just to see what happens. Some of the best learning experiences occur when students "color outside the lines."

You can download the starter files referred to in this book from the Apress Web site at `www.apress.com`. The UML modeling activities in Part 1 are for someone using Objecteering's UML Modeler. I chose this program because of its simple user interface and the fact it can be downloaded for free at `www.objecteering.com`. You do not need a CASE tool to complete these activities; a paper and pencil will work just fine. You can also use another CASE tool such as Visio to complete the activities. The activities in Part 2 require Visual Studio 2005 with Visual Basic installed. I encourage you to install the help files and make ample use of them while completing the activities. The activities in Part 3 require Microsoft SQL Server 2000 or 2005 with the Pubs and Northwind databases installed. Appendix C includes instructions on downloading and installing the sample databases. You can find a trial edition of both Visual Studio 2005 and SQL Server 2005 at `www.msdn.microsoft.com`.

■**Note** The web addresses mentioned are subject to change without notice. Check the Apress site (`www.apress.com`) for any updates.

PART 1

Object-Oriented Programming and Design Fundamentals

CHAPTER 1

Overview of Object-Oriented Programming

To set the stage for your study of object-oriented programming and Visual Basic, this chapter will briefly look at the history of object-oriented programming and the characteristics of an object-oriented programming language. You will look at why object-oriented programming has become so important in the development of industrial-strength distributed software systems. You will also examine how Visual Basic has evolved into one of the leading business application programming languages.

After reading this chapter, you will be familiar with the following:

- What object-oriented programming is
- Why object-oriented programming has become so important in the development of industrial-strength applications
- The characteristics that make a programming language object-oriented
- The history and evolution of Visual Basic

What Is OOP?

Object-oriented programming (OOP) is an approach to software development in which the structure of the software is based on *objects* interacting with each other to accomplish a task. This interaction takes the form of messages passing back and forth between the objects. In response to a message, an object can perform an action, or *method*.

If you look at how you accomplish tasks in the world around you, you can see that you interact in an object-oriented world. If you want to go to the store, for example, you interact with a car object. A car object consists of other objects that interact with each other to accomplish the task of getting you to the store. You put the key in the ignition object and turn it. This, in turn, sends a message (through an electrical signal) to the starter object, which interacts with the engine object to start the car. As a driver, you are isolated from the logic of how the objects of the system work together to start the car. You just initiate the sequence of events by executing the start method of the ignition object with the key. You then wait for a response (message) of success or failure.

Similarly, users of software programs are isolated from the logic needed to accomplish a task. For example, when you print a page in your word processor, you initiate the action by clicking a print button. You are isolated from the internal processing that needs to occur; you just wait for a response telling you if it printed. Internally, the button object interacts with a printer object, which interacts with the printer to accomplish the task of printing the page.

OOP concepts started surfacing in the mid-1960s with a programming language called Simula and further evolved in the 1970s with advent of Smalltalk. Although software developers did not overwhelmingly embrace these early advances in OOP languages, object-oriented methodologies continued to evolve. A resurgence of interest in object-oriented methodologies occurred in the mid-1980s. Specifically, OOP languages such as C++ and Eifle became popular with mainstream computer programmers. OOP continued to grow in popularity in the 1990s, most notably with the advent of Java and the huge following it attracted. And in 2002, in conjunction with the release of the .NET Framework, Microsoft introduced a new OOP language, C# (pronounced *C-sharp*) and revamped Visual Basic so that it is truly an OOP language.

Why Use OOP?

Why has OOP developed into such a widely used paradigm for solving business problems today? During the 1970s and 1980s, procedural-oriented programming languages such as C, Pascal, and Fortran were widely used to develop business-oriented software systems. Procedural languages organize the program in a linear fashion—they run from top to bottom. In other words, the program is a series of steps that run one after another. This type of programming worked fine for small programs that consisted of a few hundred code lines, but as programs became larger, they became hard to manage and debug.

In an attempt to manage the ever-increasing size of the programs, structured programming was introduced to break down the code into manageable segments called *functions* or *procedures*. This was an improvement, but as programs performed more complex business functionality and interacted with other systems, the shortcomings of structural programming methodology began to surface:

- Programs became harder to maintain.
- Existing functionality was hard to alter without adversely affecting all of the system's functionality.
- New programs were essentially built from scratch. Consequently, there was little return on the investment of previous efforts.
- Programming was not conducive to team development. Programmers needed to know every aspect of how a program worked and could not isolate their efforts on one aspect of a system.
- It was hard to translate business models into programming models.
- It worked well in isolation but did not integrate well with other systems.

In addition to these shortcomings, some evolutions of computing systems caused further strain on the structural program approach:

- Nonprogrammers demanded and were given direct access to programs through the incorporation of graphical user interfaces and their desktop computers.
- Users demanded a more-intuitive, less-structured approach to interacting with programs.
- Computer systems evolved into a distributed model where the business logic, user interface, and backend database were loosely coupled and accessed over the Internet and intranets.

As a result, many business software developers turned to object-oriented methodologies and programming languages to solve these problems. The benefits included the following:

- A more intuitive transition from business analysis models to software implementation models
- The ability to maintain and implement changes in the programs more efficiently and rapidly
- The ability to more effectively create software systems using a team process, allowing specialists to work on parts of the system
- The ability to reuse code components in other programs and purchase components written by third-party developers to increase the functionality of programs with little effort
- Better integration with loosely coupled distributed computing systems
- Improved integration with modern operating systems
- The ability to create a more intuitive graphical user interface for the users

The Characteristics of OOP

In this section, you are going to look at the some fundamental concepts and terms common to all OOP languages. Do not worry about how these concepts get implemented in any particular programming language; that will come later. My goal is to merely familiarize you with the concepts and relate them to your everyday experiences in such a way that they make more sense later when you look at OOP design and implementation.

Objects

As I noted earlier, we live in an object-oriented world. You are an object. You interact with other objects. To write this book I am interacting with a computer object. When I woke up this morning, I was responding to a message sent out by an alarm clock object. In fact, you are an object with data such as height and hair color. You also have methods that you perform or are performed on you—for example, eating and walking.

So what are objects? In OOP terms, an *object* is a structure for incorporating data and the procedures for working with that data. For example, if you were interested in tracking data associated with products in inventory, you would create a product object that is responsible for maintaining and working with the data pertaining to the products. If you wanted to have printing capabilities in your application, you would work with a printer object that is responsible for the data and methods used to interact with your printers.

Abstraction

When you interact with objects in the world, you are often concerned with only a subset of their properties. Without this ability to abstract or filter out the extraneous properties of objects, you would find it hard to process the plethora of information bombarding you and concentrate on the task at hand.

As a result of *abstraction*, when two different people interact with the same object, they often deal with a different subset of attributes. When I drive my car, for example, I need to know the speed of the car and the direction it is going. Because the car is an automatic, I do not need to know the RPMs of the engine, so I filter this information out. On the other hand, this information would be critical to a racecar driver, who would not filter it out.

When constructing objects in OOP applications, it is important to incorporate this concept of abstraction. If you were building a shipping application, you would construct a product object with attributes such as size and weight. The color of the item would be extraneous information and filtered out. On the other hand, when constructing an order-entry application, the color could be important and would be included as an attribute of the product object.

Encapsulation

Another important feature of OOP is *encapsulation*. Encapsulation is the process in which no direct access is granted to the data; instead, it is hidden. If you want to gain access to the data, you must interact with the object responsible for the data. In the previous inventory example, if you wanted to view or update information on the products, you would need to work through the product object. To read the data, you would send the product object a message. The product object would then read the value and send back a message telling you what the value is. The product object defines what operations can be performed on the product data. If you send a message to modify the data and the product object determines it is a valid request, it will perform the operation for you and send a message back with the result.

You experience encapsulation in your daily life all the time. Think about a human resources department. The human resources staff members encapsulate (hide) the information about employees. They determine how this data can be used and manipulated. Any request for the employee data or request to update the data must be routed through them. Another example is network security. Any request for the security information or a change to a security policy must be made through a network security administrator. The security data is encapsulated from the users of the network.

By encapsulating data, you make the data of your system more secure and reliable. You know how the data is being accessed and what operations are being performed on the data. This makes program maintenance much easier and also greatly simplifies the debugging process. You can also modify the methods used to work on the data, and if you do not alter how the method is requested and the type of response sent back, then you do not need to alter the other objects using the method. Think about when you send a letter in the mail. You make a request to the post office to deliver the letter. How the post office accomplishes this is not exposed to you. If it changes the route it uses to mail the letter, it does not affect how you initiate the sending of the letter. You do not need to know the post office's internal procedures used to deliver the letter.

Polymorphism

Polymorphism is the ability of two different objects to respond to the same request message in their own unique way. For example, I could train my dog to respond to the command "bark" and my bird to respond to the command "chirp." On the other hand, I could train them to both respond to the command "speak." Through polymorphism, I know that the dog will respond with a bark and the bird will respond with a chirp.

How does this relate to OOP? You can create objects that respond to the same message in their own unique implementations. For example, you could send a print message to a printer object that would print the text on a printer, and you could send the same message to a screen object that would print the text to a window on your computer screen.

Another good example of polymorphism is the use of words in the English language. Words have many different meanings, but through the context of the sentence, you can deduce which meaning is intended. You know that someone who says, "Give me a break!" is not asking you to break his leg!

In OOP, you implement this type of polymorphism through a process called *overloading*. You can implement different methods of an object that have the same name. The object can then tell which method to implement depending on the context (in other words, the number and type of arguments passed) of the message. For example, you could create two methods of an inventory object to look up the price of a product. Both of these methods would be named `getPrice`. Another object could call this method and pass either the name of the product or the product ID. The inventory object could tell which `getPrice` method to run by whether a string value or an integer value was passed with the request.

Inheritance

Most objects are classified according to hierarchies. For example, you can classify all dogs together as having certain common characteristics, such as having four legs and fur. Their breeds further classify them into subgroups with common attributes, such as size and demeanor. You also classify objects according to their function. For example, there are commercial vehicles and recreational vehicles. There are trucks and passenger cars. You classify cars according to their make and model. To make sense of the world, you need to use object hierarchies and classifications.

You use *inheritance* in OOP to classify the objects in your programs according to common characteristics and function. This makes working with the objects easier and more intuitive. It also makes programming easier, because it enables you to combine general characteristics into a parent object and inherit these characteristics in the child objects. For example, you can define an employee object that defines all the general characteristics of employees in your company. You can then define a manager object that inherits the characteristics of the employee object but also adds characteristics unique to managers in your company. The manager object will automatically reflect any changes in the implementation of the employee object.

Aggregation

Aggregation is when an object consists of a composite of other objects that work together. For example, your lawn mower object is a composite of the wheel objects, the engine object, the blade object, and so on. In fact, the engine object is a composite of many other objects. There are many examples of aggregation in the world around us. The ability to use aggregation in OOP is a powerful feature that enables you to accurately model and implement business processes in your programs.

The History of Visual Basic

By most accounts, you can trace the origins of Visual Basic to Alan Cooper, an independent software vendor. In the late 1980s, Cooper was developing a shell construction kit called Tripod. What made Tripod unique was it incorporated a visual design tool that enabled developers to design their Windows interfaces by dragging and dropping controls onto it. Using a visual design tool hid a lot of the complexity of the Windows Application Programming Interface (API) from the developer. The other innovation associated with Tripod was the extensible model it offered programmers. Programmers could develop custom controls and incorporate them into the Tripod development environment. Up to this point, development tools were, for the most part, closed environments that could not be customized.

Microsoft paid Cooper for the development of Tripod and renamed it Ruby. Although Microsoft never released Ruby as a shell construction kit, it incorporated its form engine with the QuickBasic programming language and developed Thunder, one of the first rapid application development (RAD) tools for Windows programs. Thunder was renamed to Visual Basic, and Visual Basic 1.0 was introduced in the spring of 1991.

Visual Basic 1.0 became popular with business application developers because of its ease of use and its ability to rapidly develop prototype applications. Although Visual Basic 1.0 was an innovation in the design of Windows applications, it did not have built-in support for database interactivity. Microsoft realized this was a severe limitation and introduced native support for data access in the form of Data Access Objects (DAO) in Visual Basic 3.0. After the inclusion of native data support, the popularity of Visual Basic swelled. It transitioned from being a prototyping tool to being a tool used to develop industrial-strength business applications.

Microsoft has always been committed to developing the Visual Basic language and the Visual Basic integrated development environment (IDE). In fact, by many accounts, Bill Gates himself has taken an active interest in the development and growth of Visual Basic. At one point, the design team did not allow controls to be created and added to the Toolbox. When Bill Gates saw the product demo, he insisted that this extensibility be incorporated into the product. This extensibility brought on the growth of the custom control industry.

Third-party vendors began to market controls that made programming an application even easier for Visual Basic developers. For example, one vendor marketed a Resize control that encapsulated the code needed to resize a form and the controls the form contained. A developer could purchase this tool and add it to the Toolbox in the Visual Basic IDE. The developer could then simply drag the resize control onto the form, and the form and the controls it contained would resize proportionally.

By version 6.0, Visual Basic had evolved into a robust and industrial-strength programming language with an extremely large and dedicated developer base. Nevertheless, as strong as Visual Basic had become as a programming language, many programmers felt it had one major shortcoming. They considered Visual Basic to be an *object-like* programming language—not a true object-oriented programming language. Although Visual Basic 4.0 gave developers the ability to create classes and to package the classes in reusable components, Visual Basic did not incorporate basic OOP features such as inheritance and method overloading. Without these features, developers were severely limited in their ability to construct complex distributed software systems. Microsoft recognized these shortcomings and changed Visual Basic into a true OOP language with the release of Visual Basic .NET 1.0.

Since the .NET Framework's initial release in 2002, Microsoft has continued to improve and innovate it, along with the core languages built on top of the framework: C# and Visual Basic. Microsoft is also committed to providing .NET developers with the tools necessary to have a highly productive and intuitive programming experience.

With the release of Visual Basic 2005 and Visual Studio 2005, Microsoft has greatly enhanced both the language and the design-time developing experience for Visual Basic developers. As you work your way through this book, I think you will come to appreciate the power and productivity that Visual Studio and the Visual Basic language provides.

Summary

In this chapter, you were introduced to OOP and got a brief history of Visual Basic. Now that you have an understanding of what constitutes an OOP language and why OOP languages are so important to enterprise-level application development, your next step is to become familiar with how OOP applications are designed.

Successful applications must be carefully planned and developed before any meaningful coding takes place. The next chapter is the first in a series of three aimed at introducing you to some of the techniques used when designing object-oriented applications. You will look at the process of deciding which objects need to be included in an application and which attributes of these objects are important to the functionality of that application.

CHAPTER 2

Designing OOP Solutions: Identifying the Class Structure

Most software projects you will become involved with as a business software developer will be a team effort. As a programmer on the team, you will be asked to transform the design documents into the actual application code. Additionally, because the design of object-oriented programs is a recursive process, designers depend on the feedback of the software developers to refine and modify the program design. As you gain experience in developing object-oriented software systems, you may even be asked to sit in on the design sessions and contribute to the design process. Therefore, as a software developer, you should be familiar with the purpose and the structure of the various design documents, as well as have some knowledge of how these documents are developed.

This chapter introduces you to some of the common documents used to design the static aspects of the system. (You'll learn how the dynamic aspects of the system are modeled in the next chapter.) To help you understand these documents, this chapter includes some hands-on activities, based on a limited case study. You'll find similar activities corresponding to the topics of discussion in most of the chapters in this book.

After reading this chapter, you will be familiar with the following:

- The goals of software design
- The fundamentals of the Unified Modeling Language
- The purpose of a software requirement specification
- How use case diagrams model the services the system will provide
- How class diagrams model the classes of objects that need to be developed

Goals of Software Design

A well-organized approach to system design is essential when developing modern enterprise-level object-oriented programs. The design phase is one of the most important in the software development cycle. You can trace many of the problems associated with failed software projects to poor upfront design and inadequate communication between the system's developers and the system's consumers. Unfortunately, many programmers and program managers do not like getting involved in the design aspects of the system. They view any time not spent cranking out code as unproductive.

To make matters worse, with the advent of "Internet time," consumers expect increasingly shorter development cycles. So, to meet unrealistic timelines and project scope, developers tend to forgo or cut short the system design phase of development. This is truly counterproductive to the system's success. Investing time in the design process will achieve the following:

- Provide an opportunity to review the current business process and fix any inefficiencies or flaws uncovered
- Educate the customers as to how the software development process occurs and incorporate them as partners in this process
- Create realistic project scopes and timelines for completion
- Provide a basis for determining the software testing requirements
- Reduce the cost and time required to implement the software solution

A good analogy to software design is the process of building a home. You would not expect the builder to start working on the house without detailed plans (blueprints) supplied by an architect. You would also expect the architect to talk to you about the home's design before creating the blueprints. It is the architect's job to consult with you about the design and functionality you want in the house and convert your requests to the plans that the builder uses to build the home. A good architect will also educate you as to what features are reasonable for your budget and projected timeline.

Understanding the Unified Modeling Language

To successfully design object-oriented software, you need to follow a proven design methodology. One of the most common design methodologies used in OOP today is the Unified Modeling Language (UML).

UML was developed in the early 1980s as a response to the need for a standard, systematic way of modeling the design of object-oriented software. It consists of a series of textual and graphical models of the proposed solution. These models define the system scope, components of the system, user interaction with the system, and how the system components interact with each other to implement the system functionality.

The following are some common models used in UML:

- **Software requirement specification (SRS):** A textual description of the overall responsibilities and scope of the system.
- **Use case:** A textual/graphical description of how the system will behave from the users' perspective. Users can be humans or other systems.
- **Class diagram:** A visual blueprint of the objects that will be used to construct the system.
- **Sequence diagram:** A model of the sequence of object interaction as the program executes. Emphasis is placed on the order of the interactions and how they proceed over time.
- **Collaboration diagram:** A view of how objects are organized to work together as the program executes. Emphasis is placed on the communications that occur between the objects.
- **Activity diagram:** A visual representation of the flow of execution of a process or operation.

In this chapter, you'll look at the development of the SRS, use cases, and class diagrams. The next chapter covers the sequence, collaboration, and activity diagrams.

Developing an SRS

The purpose of the SRS is to do the following:

- Define the functional requirements of the system
- Identify the boundaries of the system
- Identify the users of the system
- Describe the interactions between the system and the external users
- Establish a common language between the client and the program team for describing the system
- Provide the basis for modeling use cases

To produce the SRS, you interview the business owners and the end users of the system. The goals of these interviews are to clearly document the business processes involved and establish the system's scope. The outcome of this process is a formal document (the SRS) detailing the functional requirements of the system. A formal document helps to ensure agreement between the customers and the software developers. The SRS also provides a basis for resolving any disagreements over "perceived" system scope as development proceeds.

As an example, suppose that the owners of a small commuter airline want customers to be able to view flight information and reserve tickets for flights using a web registration system. After interviewing the business managers and the ticketing agents, the software designers draft an SRS document that lists the system's functional requirements. The following are some of these requirements:

- Nonregistered web users can browse to the web site to view flight information, but they cannot book flights.
- New customers wanting to book flights must complete a registration form providing their name, address, company name, phone number, fax number, and e-mail address.
- A customer is classified as either a corporate customer or a retail customer.
- Customers can search for flights based on destination and departure times.
- Customers can book flights indicating the flight number and the number of seats requested.
- The system sends customers a confirmation via e-mail when the flight is booked.
- Corporate customers receive frequent flier miles when their employees book flights. Frequent-flier miles are used to discount future purchases.
- Ticket reservations can be canceled up to one week in advance for an 80-percent refund.
- Ticketing agents can view and update flight information.

In this partial SRS document, you can see that several succinct statements define the system scope. They describe the functionality of the system as viewed by the system's users and identify the external entities that will use it. It is important to note that the SRS does *not* contain references to the technical requirements of the system.

Once the SRS is developed, the functional requirements it contains are transformed into a series of use case diagrams.

Introducing Use Cases

Use cases describe how external entities will use the system. These external entities can be either humans or other systems and are referred to as *actors* in UML terminology. The description emphasizes the users' view of the system and the interaction between the users and the system. Use cases help to further define system scope and boundaries. They are usually in the form of a diagram, along with a textual description of the interaction taking place. Figure 2-1 shows a generic diagram that consists of two actors represented by stick figures, the system represented by a rectangle, and use cases depicted by ovals inside the system boundaries.

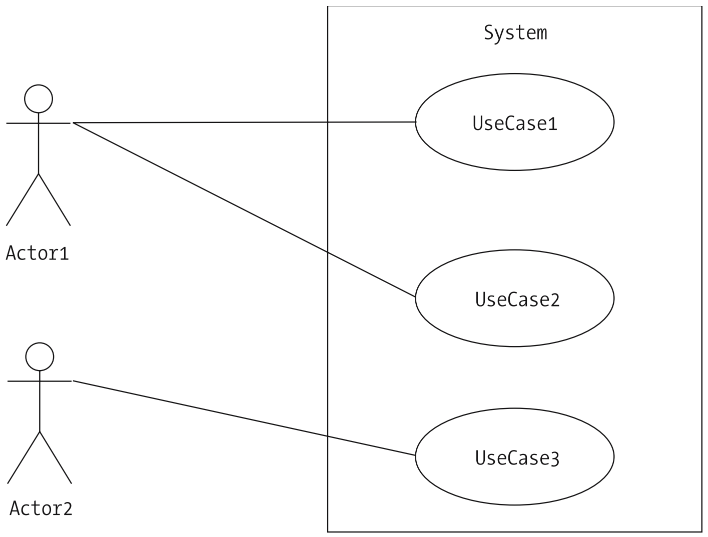

Figure 2-1. *Generic use case diagram with two actors and three use cases*

Use cases are developed from the SRS document. The actor is any outside entity that interacts with the system. An actor could be a human user (for instance, a rental agent), another software system (for instance, a software billing system), or an interface device (for instance, a temperature probe). Each interaction that occurs between an actor and the system is modeled as a use case.

The sample use case shown in Figure 2-2 was developed for the flight booking application introduced in the previous section. It shows the use case diagram for the requirement "Customers can search for flights based on destination and departure times."

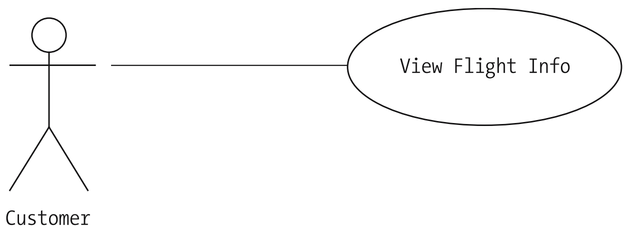

Figure 2-2. *View Flight Info use case diagram*

Along with the graphical depiction, many designers and software developers find it helpful to provide a textual description of the use case. The textual description should be succinct and focused on *what* is happening and not on *how* it is occurring. Sometimes, any preconditions or postconditions associated with the use case are also identified. The following text further describes the use case diagram shown in Figure 2-2:

- **Description:** A customer views the flight information page. The customer enters flight search information. After submitting the search request, the customer views a list of flights matching the search criteria.
- **Preconditions:** None.
- **Postconditions:** The customer has the opportunity to log in and proceed to the flight booking page.

As another example, take a look at the Reserve Seat use case shown in Figure 2-3.

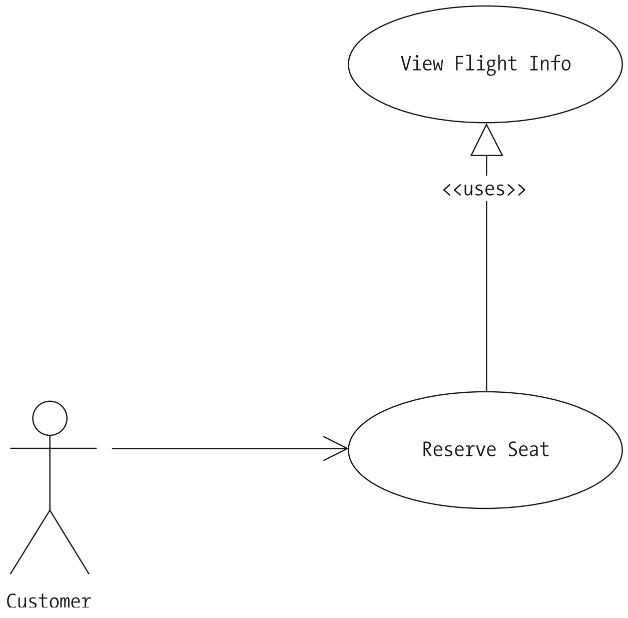

Figure 2-3. *Reserve Seat use case diagram*

The following text further describes the use case diagram shown in Figure 2-3:

- **Description:** The customer enters the flight number and indicates the seats being requested. After the customer submits the request, confirmation information is displayed.
- **Preconditions:** The customer has looked up the flight information. The customer has logged in and is viewing the flight booking screen.
- **Postconditions:** The customer is sent a confirmation e-mail outlining the flight details and the cancellation policy.

As you can see from Figure 2-3, certain relationships can exist between use cases. The Reserve Seat use case includes the View Flight Info use case. This relationship is useful because you can use the View Flight Info use case independently of the Reserve Flight use case. This is called *inclusion*. You cannot use the Reserve Seat use case independently of the View Flight Info use case, however. This is important information that will affect how you model the solution.

Another way that use cases relate to each other is through *extension*. You might have a general use case that is the base for other use cases. The base use case is extended by other use cases. For example, you might have a Register Customer use case that describes the core process of registering customers. You could then develop Register Corporate Customer and Register Retail Customer use cases that extend the base use case. The difference between extension and inclusion is that in extension, the base use case being extended is not used on its own. Figure 2-4 demonstrates how you model extension in a use case diagram.

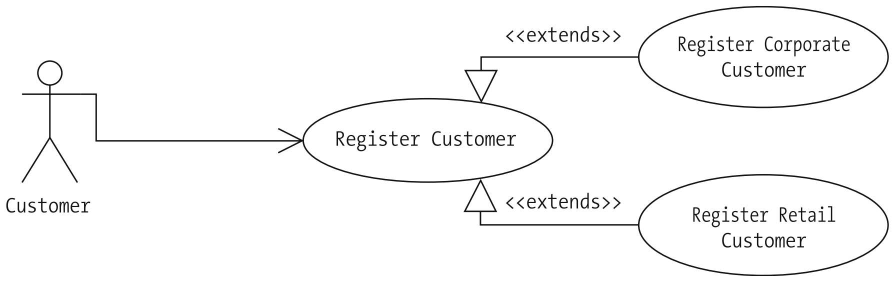

Figure 2-4. *Extending use cases*

A common mistake when developing use cases is to include actions initiated by the system itself. The emphasis of the use case is on the interaction between external entities and the system. Another common mistake is to include a description of the technical requirements of the system. Remember that use cases do not focus on how the system will perform the functions, but rather on what functions need to be incorporated in the system from the users' standpoint.

After you have developed the use cases of the system, you can begin to identify the internal system objects that will carry out the system's functional requirements. You do this through the use of a class diagram.