Developing Robust & Scalable Objects with OOP

Object-Oriented Programming

with Visual Basic .NET

O'REILLY®

J.P. Hamilton

Object-Oriented Programming with Visual Basic .NET

J.P. Hamilton

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

CHAPTER 1

Introduction

To understand the world of object-oriented programming, look at the world around you for a moment. You might see vacuum cleaners, coffee makers, ceiling fans, and a host of other objects. Everywhere you look, objects surround you.

Some of these objects, such as cameras, operate independently. Some, such as telephones and answering machines, interact with one another. Some objects contain data that persists between uses, like the address book in a cell phone. Some objects contain other objects, like an icemaker inside of the freezer.

Many objects are similar in function but different in purpose. Bathtubs and kitchen sinks, for example, both provide water and are used for cleaning. But it is a rare occasion when you will take a bath in the kitchen sink or wash your dishes in the tub. However, the bathtub and the kitchen sink in your house probably share the same plumbing. Certainly, they share a common interface: hot and cold water knobs, a faucet, and a drain.

When you think about it, what is the difference between a sink and a bathtub? The location? The size of the basin? Their heights off the ground? How many more similarities are there than differences?

Sometimes the same action causes an object to do different things depending on the context of the situation. When you press Play on the remote, the DVD might play a movie on the television. But if a CD is in the player, it plays music out of the speakers. Same button, same action—different results. When you flip the switch on the back porch, the light comes on. But the switch in the kitchen turns on the garbage disposal. You use the same kind of switch, but obtain different results.

You can think about many objects around you in terms of *black boxes*. You comprehend the fundamentals of these objects and possess a basic understanding of what makes them work, but the specifics of their operation are unknown to you. And you like it that way. Do you really want to have to know the inner mechanisms of every object in your house in order to use it?

Consider that light bulb on the back porch. The filament in the bulb is nothing more than a simple resistor. When the 100-watt bulb is "on," the filament's temperature is about 2550 degrees Celsius. The resulting thermal radiation, which is proportional to the length of the filament (but not the diameter), produces about 1750 lumens worth of visible light at a wavelength of about 555 nanometers. And by the way, the filament is made out of tungsten.

Do you really want to know these minute details, or do you just want the light to come on when you flick the switch?

Any object has two inherent properties: *state* and *behavior*. The light bulb on the back porch has state. It can be on or off. It has a brand name and a life expectancy. It has been in use for a certain number of hours. It has a specified number of hours left before the irregular evaporation of its tungsten filament causes it to burn out. Behaviorally, it provides light; it shines.

But an object is rarely an island unto itself.

Many objects participate collectively in a system. The television and surrounding sound speakers are a part of a system called a home theater. The refrigerator and oven belong to a system called a kitchen. These systems, in turn, are a part of a larger system that is called an apartment. Collections of apartments make up a system known as a complex. Apartments and houses belong to neighborhoods and so on, ad infinitum.

In essence, this book discusses systems. Building and designing objects is one aspect of this process of building a system. Determining how these objects interact with one another is another. Understanding both phases of development is crucial when building any system that has more than a modicum of complexity.

Generally, you can think of this process of developing a system as object-oriented programming and object-oriented design. Specifically, though, you are really working toward an understanding of the objects you build and the system in which they participate. Component-based programming forms the basis of this system.

Programming objects in software doesn't require an object-oriented language, and just because you use an object-oriented programming language doesn't mean that your code is object-oriented. Languages can only assist the process; they can't make any guarantees. The ability to write object-oriented software was always available with VB. Writing it just hasn't always been easy because the language wasn't always oriented in that direction. Developing binary reusable components in VB has been possible for some time now, but using these components across languages used to be considered somewhat of a black art—until now.

Today, Visual Basic .NET is a cutting-edge, object-oriented language that runs inside of a state-of-the-art environment. It is feature-rich and designed to take advantage of the latest developments in object-oriented programming. Writing software and building components has never been easier.

Visual Basic .NET and Object-Oriented Programming

Visual Basic .NET is a fully object-oriented programming language, which means it supports the four basic tenets of object-oriented programming: abstraction, encapsulation, inheritance, and polymorphism.

We have already conceptualized many of these object-oriented concepts by just looking at the objects that surround us in our everyday lives. Let's look more closely at these terms and see what they actually mean and what they do for developers of object-oriented software.

Abstraction

A radio has a tuner, an antenna, a volume control, and an on/off switch. To use it, you don't need to know that the antenna captures radio frequency signals, converts them to electrical signals, and then boosts their strength via a high-frequency amplification circuit. Nor do you need to know how the resulting current is filtered, boosted, and finally converted into sound. You merely turn on the radio, tune in the desired station, and listen. The intrinsic details are invisible. This feature is great because now everyone can use a radio, not just people with technical know-how. Hiring a consultant to come to your home every time you wanted to listen to the radio would become awfully expensive. In other words, you can say that the radio is an object that was designed to hide its complexity.

If you write a piece of software to track payroll information, you would probably want to create an Employee object. People come in all shapes, sizes, and colors. They have different backgrounds, enjoy different hobbies, and have a multitude of beliefs. But perhaps, in terms of the payroll application, an employee is just a name, a rank, and a serial number, while the other qualities are not relevant to the application. Determining what something is, in terms of software, is *abstraction*.

In object-oriented software, complexity is managed by using abstraction. Abstraction is a process that involves identifying the crucial behavior of an object and eliminating irrelevant and tedious details. A well thought-out abstraction is usually simple, slanted toward the perspective of the user (the developer using your objects), and has probably gone through several iterations. Rarely is the initial attempt at an abstraction the best choice.

Remember that the abstraction process is context sensitive. In an application that will play music, the radio abstraction will be completely different from the radio abstraction in a program designed to teach basic electronics. The internal details of the latter would be much more important than the former.

Encapsulation

Programming languages like C and Pascal can both produce object-like constructs. In C, this feature is called a *struct*; in Pascal, it is referred to as a *record*. Both are user-defined data types. In both languages, a function can operate on more than one data type. The inverse is also true: more than one function can operate on a single data type. The data is fully exposed and vulnerable to the whims of anyone who has an instance of the type because these languages do not explicitly tie together data and the functions that operate on that data.

In contrast, object-oriented programming is based on *encapsulation*. When an object's state and behavior are kept together, they are encapsulated. That is, the data that represents the state of the object and the methods (Functions and Subs) that manipulate that data are stored together as a cohesive unit.

Encapsulation is often referred to as *information hiding*. But although the two terms are often used interchangeably, information hiding is really the result of encapsulation, not a synonym for it. They are distinct concepts. Encapsulation makes it possible to separate an object's implementation from its behavior—to restrict access to its internal data. This restriction allows certain details of an object's behavior to be hidden. It allows us to create a "black box" and protects an object's internal state from corruption by its clients.

Encapsulation is also frequently confused with abstraction. Though the two concepts are closely related, they represent different ideas. Abstraction is a process. It is the act of identifying the relevant qualities and behaviors an object should possess. Encapsulation is the mechanism by which the abstraction is implemented. It is the result. The radio, for instance, is an object that encapsulates many technologies that might not be understood clearly by most people who benefit from it.

In Visual Basic .NET, the construct used to define an abstraction is called a *class*. The terms *class* and *object* are often used interchangeably, but an object is actually an instance of a class. A component is a collection of one or more object definitions, like a class library in a DLL.

Inheritance

Inheritance is the ability to define a new class that inherits the behaviors (and code) of an existing class. The new class is called a *child* or *derived class*, while the original class is often referred to as the *parent* or *base class*.

Inheritance is used to express "is-a" or "kind-of" relationships. A car *is a* vehicle. A boat *is a* vehicle. A submarine *is a* vehicle. In OOP, the `Vehicle` *base class* would provide the common behaviors of all types of vehicles and perhaps delineate behaviors all vehicles must support. The particular *subclasses* (i.e., derived classes) of vehicles would implement behaviors specific to that type of vehicle. The main concepts behind inheritance are extensibility and code reuse.

In contrast to inheritance, there is also the notion of a "has-a" relationship. This relationship is created by using *composition*. Composition, which is sometimes referred to as aggregation, means that one object contains another object, rather than inheriting an object's attributes and behaviors. Naturally, a car *has an* engine, but it is not a *kind of* engine.

C++ supports a type of reuse called *multiple inheritance*. In this scenario, one class inherits from more than one base class. But many C++ programmers will tell you that using multiple inheritance can be tricky. Base classes with identical function names or common base classes can create nightmares for even the most experienced programmers.

VB.NET, like Java, avoids this problem altogether by providing support only for single inheritance. But don't worry, you aren't missing out on anything. Situations that seem ideal for multiple inheritance can usually be solved with composition or by rethinking the design.

When it comes to proper object-oriented design, a deep understanding of inheritance and its effects is crucial. Deriving new classes from existing classes is not always as straightforward as it might initially appear. Is a circle a kind of ellipse? Is a square a kind of rectangle? Mistakes in an inheritance hierarchy can cripple an object model.

Polymorphism

Polymorphism refers to the ability to assume different forms. In OOP, it indicates a language's ability to handle objects differently based on their runtime type.

When objects communicate with one another, we say that they *send* and *receive* messages. The advantage of polymorphism is that the sender of a message doesn't need to know which class the receiver is a member of. It can be any arbitrary class. The sending object only needs to be aware that the receiving object can perform a particular behavior.

A classic example of polymorphism can be demonstrated with geometric shapes. Suppose we have a `Triangle`, a `Square`, and a `Circle`. Each class *is a* `Shape` and each has a method named `Draw` that is responsible for rendering the `Shape` to the screen.

With polymorphism, you can write a method that takes a `Shape` object or an array of `Shape` objects as a parameter (as opposed to a specific kind of `Shape`). We can pass `Triangles`, `Circles`, and `Squares` to these methods without any problems, because referring to a class through its parent is perfectly legal. In this instance, the receiver is only aware that it is getting a `Shape` that has a method named `Draw`, but it is ignorant of the specific kind of `Shape`. If the `Shape` were a `Triangle`, then `Triangle`'s version of `Draw` would be called. If it were a `Square`, then `Square`'s version would be called, and so on.

We can illustrate this concept with a simple example. Suppose we are working on a small graphics package and we need to draw several shapes on the screen at one time. To implement this functionality, we create a class called `Scene`. `Scene` has a method named `Render` that takes an array of `Shape` objects as a parameter. We can now create an array of different kinds of shapes and pass it to the `Render` method. `Render` can iterate through the array and call `Draw` for each element of the array, and the appropriate version of `Draw` will be called. `Render` has no idea what specific kind of `Shape` it is dealing with.

The big advantage to this implementation of the `Scene` class and its `Render` method is that two months from now, when you want to add an `Ellipse` class to your graphics package, you don't have to touch one line of code in the `Scene` class. The `Render` method can draw an `Ellipse` just like any other `Shape` because it deals with them generically. In this way, the `Shape` and `Scene` classes are *loosely coupled*, which is something you should strive for in a good object-oriented design.

This type of polymorphism is called *parametric polymorphism*, or *generics*. Another type of polymorphism is called *overloading*. Overloading occurs when an object has two or more behaviors that have the same name. The methods are distinguished only by the messages they receive (that is, by the parameters of the method).

Polymorphism is a very powerful concept that allows the design of amazingly flexible applications. Chapter 4 discusses polymorphism in more depth.

The .NET Framework

The objects you construct with VB.NET will live out their lives within the .NET Framework, which is a platform used to develop applications. The platform was designed from the ground up by using open standards and protocols like XML, HTTP, and SOAP. It contains a rich standard library that provides services available to any language running under its protection.

The impetus behind its creation was the desire to develop a platform for building, deploying, and running web-based services. In spite of this goal, the framework is ideal for developing all types of applications, regardless of the design. The .NET Framework makes child's play of some of programming's most sophisticated concepts, giving you the ability to take advantage of today's cutting-edge architectures:

- Distributed computing using open Internet standards and protocols such as HTTP, XML, and SOAP
- Enterprise services such as object pooling, messaging, security, and transactions
- An infrastructure that simplifies the development of reusable cross-language compatible components that can be deployed over the Internet
- Simplified web development using open standards
- Full language integration that make it possible to inherit from classes, catch exceptions, and debug across different languages

Deployment is made simpler because settings are stored in XML-based configuration files that reside in the application directory; there is no need to go to the registry. Shared DLLs must have a unique hash value, locale, and version, so physical filenames are no longer important once these considerations are met. Not having physical filenames makes it possible to have several different versions of the same DLL in use at the same time, which is known as *side-by-side* execution. All dependencies and references are stored within the executable in a section called the *manifest*. In a sense, we're back to the days of DOS because to deploy an application, you only need to *xcopy* it from one directory to another.

This book explores many aspects of .NET in order to gain a complete understanding of the components you write and the world in which they live. Doing it any other way is impossible. The .NET Framework provides so many services your components will use that discussing one without referring to the other is literally impossible—they are that closely tied together.

Two major elements of the .NET Framework will be addressed repeatedly throughout this book. The first is the Common Language Runtime (CLR), which provides runtime services for components running under .NET.

The second element is the .NET class library, a vast toolbox containing classes for everything from data access, GUI design, and security to multithreading, networking, and messaging. The library also contains definitions for all primary data types, such as bytes, integers, and strings. All of these types are inherently derived from a base class called `System.Object`, which you can think of as a "universal" data type; there is no distinction between the types defined by the system and the types you create by writing classes or structures. Everything is an object!

The term .NET means many things to many different people. When the term is used in this book, it always refers to the .NET Framework—the Common Language Runtime and the .NET class library.

In the past, passing a string from a component written in VB to one written in C++ (or vice versa) could be frustrating. Strings in VB weren't the same as the strings in C++. In fact, under some circumstances, using a component written in C++ from VB was downright impossible because of issues involving data types. VB just doesn't know what to do with an `LPSTR`! Every language under .NET uses the same data types defined in the base class library, so interoperability problems of the past are no longer an issue.

This book touches on several major areas of the library and focuses on the development of components using VB.NET. However, if you follow the examples, you might be surprised at just how much you know.

The Common Language Runtime

The CLR is the execution engine for the .NET Framework. This runtime manages all code compiled with VB.NET. In fact, code compiled to run under .NET is called *managed* code to distinguish it from code running outside of the framework.

Besides being responsible for application loading and execution, the CLR provides services that will benefit component developers:

- Invocation and termination of threads and processes
- Object lifetime and memory management
- Cross-language integration
- Code access and role-based security
- Exception handling (even across languages)
- Deployment and versioning
- Interoperation between managed and unmanaged code
- Debugging and profiling support (even across languages)

Runtimes are nothing new. Visual Basic has always had some form of a runtime. Visual C++ has a runtime called *MSVCRT.DLL*. Perl, Python, and SmallTalk also use runtimes. The difference between these runtimes and the CLR is that the CLR is designed to work with multiple programming languages. Every language whose compiler targets the .NET Framework benefits from the services of the CLR as much as any other language.

.NET is also similar to Java. Java uses a runtime called the Java Virtual Machine. It can run only with Java code, so it has the same limitations as the other languages mentioned previously. Another distinction is that the JVM is an interpreter. Although all languages in the .NET environment are initially compiled to a CPU-independent language called Intermediate Language (which is analogous to Java byte code), IL is not interpreted at runtime like Java. When code is initially executed, one of several just-in-time (JIT) compilers translate the IL to native code on a method-by-method basis.

Cross-language integration is one of the major benefits provided by the CLR. If a colleague has written a base class in C#, you can define a class in VB.NET that derives from it. This is known as *cross-language inheritance*. Also, objects written in different languages can easily interoperate. The two parts of the CLR that make this interoperation possible are the Common Type System and the Common Language Specification.

Common Type System

The Common Type System (CTS) defines rules that a language must adhere to in order to participate in the .NET Framework. It also defines a set of common types and operations that exist across most programming languages and specifies how these types are used and managed within the CLR, how objects expose their functionality, and how they interoperate. The CTS forms the foundation that enables cross-language integration within .NET.

Common Language Specification

The Common Language Specification (CLS) is a subset of the CTS that describes the basic qualities used by a wide variety of languages. Components that use only the features of the CLS are said to be CLS-compliant. As a result, these components are guaranteed to be accessible from any other programming language that targets .NET. Because VB.NET is a CLS-compliant language, any class, object, or component that you build will be available from any other CLS-compliant programming language in .NET.

A First-Class Citizen

VB has always been easy to learn, but the power of simplicity came with a price. The language itself has never gotten the respect it deserves because it always hid so much from the developer; getting under the hood required a sledgehammer. This is no longer true. While VB is still a great language and is relatively painless to learn and use, you are no longer restricted in how "low you can go."

One of the most important concepts behind .NET is that all languages are on a level playing field; the choice of language should be determined more by your style than anything else. This is probably the reason why you prefer VB over other languages: you like the syntax of Visual Basic and appreciate its simplicity. No longer is choice of language a concern, because VB.NET is just as fast as C# and it does a few things, such as event declaration and conditional exception handling, better. But for the most part, any language that runs under .NET will provide you with the tools to develop cutting edge software. Thus, it truly is a matter of style. VB.NET is no more or no less of a language than any other in the .NET Framework.