

Expert VB 2008 Business Objects

Copyright © 2009 by Rockford Lhotka

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-4302-1638-4

ISBN-13 (electronic): 978-1-4302-1639-1

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

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Editor: Matthew Moodie

Technical Reviewers: James Miller, Andrés Villanueva

Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell,

Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Duncan Parkes,

Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Project Manager: Richard Dal Porto

Copy Editor: Kim Wimpsett

Associate Production Director: Kari Brooks-Copony

Production Editor: Laura Cheu

Compositor: Susan Glinert Stevens

Proofreader: Kim Burton

Indexer: John Collin

Artist: April Milne

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 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit <http://www.apress.com>.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales—eBook Licensing web page at <http://www.apress.com/info/bulksales>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com>.



Distributed Architecture

Object-oriented design and programming are big topics—entire books are devoted solely to the process of object-oriented design or to using object-oriented programming in various languages and on various programming platforms. My focus in this book isn't to teach the basics of object-oriented design or programming but rather to show how you may apply them to the creation of distributed .NET applications.

It can be difficult to apply object-oriented design and programming effectively in a physically distributed environment. This chapter is intended to provide a good understanding of the key issues surrounding distributed computing as it relates to object-oriented development. I'll cover a number of topics, including the following:

- How logical n-layer architectures help address reuse and maintainability
- How physical n-tier architectures impact performance, scalability, security, and fault tolerance
- The difference between data-centric and object-oriented application models
- How object-oriented models help increase code reuse and application maintainability
- The effective use of objects in a distributed environment, including the concepts of anchored and mobile objects
- The relationship between an architecture and a framework

This chapter provides an introduction to the concepts and issues surrounding distributed object-oriented architecture. Then, throughout this book, I'll be exploring an n-layer architecture that may be physically distributed across multiple machines. I'll show how to use object-oriented design and programming techniques to implement a framework supporting this architecture. I'll create a sample application that demonstrates how the architecture and the framework support development efforts.

Logical and Physical Architecture

In today's world, an object-oriented application must be designed to work in a variety of physical configurations. Even the term *application* has become increasingly blurry because of all the hype around service-oriented architecture (SOA). If you aren't careful, you can end up building applications by combining several applications, which is obviously confusing.

When I use the term *application* in this book, I'm referring to a set of code, objects, or components that's considered to be part of a single, logical unit. Even if parts of the application are in different .NET assemblies or installed on different machines, all the code will be viewed as being part of a singular application.

This definition works well when describing most traditional application models, such as single-tier or 2-tier rich client applications, n-tier smart client applications, web applications, and so forth.

In all those cases, the application consists of a set of objects or components that are designed to work together within the context of the application.

You can contrast this with an SOA model, where multiple services (each essentially a separate application) interact through message-based communication. In an SOA model, the idea is to build an enterprise system that is composed of applications and services. In this context, both applications and services are stand-alone, autonomous units of functionality, which means they both meet the definition of an application. Confusingly enough, this means a service is merely an application that has an XML interface instead of an HTML or graphical interface.

If you're thinking about service-oriented systems as you read this book, the term *application* means one of two things. First, it may refer to a service implementation. Second, it may refer to an application on the edge of the system that allows users to interact with the system. Edge applications are much like traditional applications, except they typically interact with services instead of databases for retrieving and storing data.

You can contrast the traditional and SOA models with a workflow model, which you're likely to encounter when using Windows Workflow Foundation (WF). In this environment, an application is often implemented (in whole or part) in the form of a workflow. However, the workflow itself merely orchestrates a set of *activities*, and each activity should be an autonomous, stand-alone unit of functionality. This means that an activity must meet the definition of an application. An activity is merely an application that has no real user interface beyond the data binding infrastructure built into WF.

Traditional, service-oriented applications and workflow applications *might* run on a single machine. However, it's very likely that they will run on multiple machines, such as a web server or a smart client and an application server. Given these varied physical environments, you're faced with the following questions:

- Where do the objects reside?
- Are the objects designed to maintain state, or should they be stateless?
- How is object-relational mapping handled when retrieving or storing data in the database?
- How are database transactions managed?

Before discussing some answers to these questions, it's important that you fully understand the difference between a *physical architecture* and a *logical architecture*. After defining these terms, I'll define objects and mobile objects and show you how they fit into the architectural discussion.

When most people talk about n-tier applications, they're talking about physical models in which the application is spread across multiple machines with different functions: a client, a web server, an application server, a database server, and so on. And this isn't a misconception—these are indeed n-tier systems. The problem is that many people tend to assume there's a one-to-one relationship between the layers (tiers) in a logical model and the tiers in a physical model, when in fact that's not always true.

A *physical* n-tier architecture is quite different from a *logical* n-layer architecture. An n-layer architecture has nothing to do with the number of machines or network hops involved in running the application. Rather, a logical architecture is all about separating different types of functionality. The most common logical separation is into an Interface layer, a Business layer, and a Data layer. These layers may exist on a single machine or on three separate machines—the logical architecture doesn't define those details.

Note There is a relationship between an application's logical and physical architectures: the logical architecture always has at least as many layers as the physical architecture has tiers. There may be more logical layers than physical tiers (because one physical tier can contain several logical layers), but never fewer.

The sad reality is that many applications have no clearly defined logical architecture. Often the logical architecture merely defaults to the number of physical tiers. This lack of a formal, logical design causes problems because it reduces flexibility. If a system is designed to operate in two or three physical tiers, then changing the number of physical tiers at a later date is typically very difficult. However, if you start by creating a logical architecture of three layers, you can switch more easily between one, two, or three physical tiers later.

Additionally, having clean separation between these layers makes your application more maintainable, because changing one layer often has minimal impact on the other layers. Nowhere is this more true than with the Interface layer (sometimes called the UI or Presentation layer), where the ability to switch between Windows Presentation Foundation (WPF), Windows Forms, Web Forms, ASP.NET MVC, and workflow and service-based interfaces is critical.

The flexibility to choose your physical architecture is important because the benefits gained by employing a physical n-tier architecture are different from those gained by employing a logical n-layer architecture. A properly designed logical n-layer architecture provides the following benefits:

- Logically organized code
- Easier maintenance
- Better reuse of code
- Better team-development experience
- Higher clarity in coding

On the other hand, a properly chosen physical n-tier architecture can provide the following benefits:

- Performance
- Scalability
- Fault tolerance
- Security

It goes almost without saying that if the physical or logical architecture of an application is designed poorly, there will be a risk of damaging the things that would have been improved had the job been done well.

N-Tier and SOA

It is important to realize that a physical service-oriented architecture is *not* the same as an n-tier architecture. In fact, the two concepts can be complementary. It is also important to know that the concept of a *logical n-layer* architecture is the same in SOA as in any other type of application model.

In logical n-layer models, a service should have the same layers as any other application: Interface, Business, and Data. In a logical n-layer model, the Interface layer consists of XML messages, but that's not a lot different from the HTML used in a web-based Interface layer. The Business layer is much the same as in any other application; it contains the business logic and behaviors that make the service useful. The Data layer is also much the same as in any other application in that it stores and retrieves data as necessary.

However, the physical n-tier model might not appear to translate to the SOA world at all. Some people would say that SOA totally obsoletes n-tier concepts, but I disagree. SOA has an important set of goals around loose coupling, reuse of functionality, and open communication. An n-tier client/server architecture has a complementary set of goals around performance, avoiding duplication of code, and targeted functionality. The reality is that *both models are useful*; in fact, they complement each other.

For example, you might use a service-oriented model to create a service that is available on the Internet. However, the service *implementation* might be n-tier, with the service interface on the web server and parts of the business implementation running on a separate application server. The result is a reusable service that enjoys high performance and security and avoids duplication of code.

Complexity

Experienced designers and developers often view a good n-tier architecture as a way of simplifying an application and reducing complexity, but this isn't necessarily the case. It's important to recognize that n-tier designs are typically *more* complex than single-tier designs. Even novice developers can visualize the design of a form or a page that retrieves data from a file and displays it to the user, but novice developers often struggle with 2-tier designs and are hopelessly lost in an n-tier environment.

With sufficient experience, architects and developers do typically find that the organization and structure of an n-tier model reduces complexity for large applications. However, even a veteran n-tier developer will often find it easier to avoid n-tier models when creating a simple form to display some simple data.

The point here is that n-tier architectures simplify the process only for large applications or complex environments. They can easily complicate matters if all you're trying to do is create a small application with a few forms that will be running on someone's desktop computer. (Of course, if that desktop computer is one of hundreds or thousands in a global organization, then the *environment* may be so complex that an n-tier solution provides simplicity.)

In short, n-tier architectures help to decrease or manage complexity when *any* of these are true:

- The application is large or complex.
- The application is one of many similar or related applications that, *when combined*, may be large or complex.
- The environment (including deployment, support, and other factors) is large or complex.

On the other hand, n-tier architectures can increase complexity when *all* of these are true:

- The application is small or relatively simple.
- The application isn't part of a larger group of enterprise applications that are similar or related.
- The environment isn't complex.

Something to remember is that even a small application is likely to grow, and even a simple environment often becomes more complex over time. The more successful your application, the more likely that one or both of these will happen. If you find yourself on the edge of choosing an n-tier solution, it's typically best to go with it. You should expect and plan for growth.

This discussion illustrates why n-tier applications are viewed as relatively complex. A lot of factors—both technical and nontechnical—must be taken into account. Unfortunately, it isn't possible to say definitively when n-tier does and doesn't fit. In the end, it's a judgment call that you, as an application architect, must make, based on the factors that affect your particular organization, environment, and development team.

Relationship Between Logical and Physical Models

Some architectures attempt to merge logical n-layer and physical n-tier concepts. Such mergers seem attractive because they seem simpler and more straightforward, but typically they aren't good in practice—they can lead people to design applications using a logical or physical architecture that isn't best suited to their needs.

The Logical Model

When you're creating an application, it's important to start with a logical architecture that clarifies the roles of all components, separates functionality so that a team can work together effectively, and simplifies overall maintenance of the system. The logical architecture must also include enough layers so that you have flexibility in choosing a physical architecture later.

Traditionally, you would devise at least a 3-layer logical model that separates the interface, the business logic, and the data-management portions of the application. Today that's rarely sufficient, because the "interface" layer is often physically split into two parts (browser and web server), and the "logic" layer is often physically split between a client or web server and an application server. Additionally, various application models have been used to break the traditional Business layer into multiple parts—model-view-controller (MVC) and facade-data-logic being two of the most popular at the moment.

This means that the logical layers are governed by the following rules:

- The logical architecture includes layers in order to organize components into discrete roles.
- The logical architecture must have at least as many layers as the anticipated physical deployment will have tiers.

Following these rules, most modern applications have four to six logical layers. As you'll see, the architecture used in this book includes five logical layers.

Cross-Layer Communication

Just because an application is organized into layers doesn't mean those layers can be deployed arbitrarily on different tiers. The code in one layer communicates with the layer immediately above or below it in the architecture. If you don't design that communication properly, it may be impossible to put a network (tier) boundary between the layers.

For example, the boundary between the Business layer and the Data layer is often highly optimized. Most applications have a network boundary between the Data layer and the rest of the application, so modern data access technologies are good at optimizing cross-network communication in this scenario.

The boundary between the Interface layer and the Business layer is often not optimized for this purpose. Many applications make use of data binding, which is a "chatty" technology involving many property, method, and event calls between these two layers. The result is that it is often impractical and undesirable to put a network boundary between these layers.

Not all layer boundaries should be designed to enable a tier boundary. You should design an architecture up front to enable the potential for tier boundaries in certain locations and to disallow them in other cases. If done properly, the result is a balance between flexibility and capability.

The Physical Model

By ensuring that the logical model has enough layers to provide flexibility, you can configure your application into an appropriate physical architecture that will depend on your performance, scalability, fault tolerance, and security requirements. The more physical tiers included, the worse the performance will be; however, there is the potential to increase scalability, security, and/or fault tolerance.

Performance and Scalability

The more physical tiers there are, the *worse* the performance? That doesn't sound right, but if you think it through, it makes perfect sense: *performance* is the speed at which an application responds to a user. This is different from *scalability*, which is a measure of how performance changes as load (such as increased users) is added to an application. To get optimal performance—that is, the fastest

possible response time for a given user—the ideal solution is to put the client, the logic, and the data on the user's machine. This means no network hops, no network latency, and no contention with other users.

If you decide that you need to support multiple users, you might consider putting application data on a central file server. (This is typical with Access and dBASE systems, for example.) However, this immediately affects performance because of contention on the data file. Furthermore, data access now takes place across the network, which means you've introduced network latency and network contention, too. To overcome this problem, you could put the data into a managed environment such as SQL Server or Oracle. This will help reduce data contention, but you're still stuck with the network latency and contention problems. Although improved, performance for a given user is still nowhere near what it was when everything ran directly on that user's computer.

Even with a central database server, scalability is limited. Clients are still in contention for the resources of the server, with each client opening and closing connections, doing queries and updates, and constantly demanding the CPU, memory, and disk resources that other clients are using. You can reduce this load by shifting some of the work to another server. An *application server*, possibly running Enterprise Services or Internet Information Services (IIS), can provide database connection pooling to minimize the number of database connections that are opened and closed. It can also perform some data processing, filtering, and even caching to offload some work from the database server.

Note It is important to realize that modern database servers can often easily handle hundreds of concurrent users in a 2-tier architecture. For most applications, scalability is *not* a good reason to move from a 2- to 3-tier model.

These additional steps provide a dramatic boost to scalability, but again at the cost of performance. The user's request now has *two* network hops, potentially resulting in double the network latency and contention. For a single user, the system gets slower; however, it is able to handle many times more users with acceptable performance levels.

In the end, the application is constrained by the most limiting resource. This is typically the speed of transferring data across the network—but if the database or application server is underpowered, it can become so slow that data transfer across the network won't be an issue. Likewise, if the application does extremely intense calculations and the client machines are slow, then the cost of transferring the data across the network to a relatively idle high-speed server can make sense.

Security

Security is a broad and complex topic, but by narrowing the discussion solely to consider how it's affected by physical n-tier decisions, it becomes more approachable. The discussion is no longer about authentication or authorization as much as it is about controlling physical access to the machines on which portions of the application will run. The number of physical tiers in an application has no impact on whether users can be authenticated or authorized, but physical tiers *can* be used to increase or decrease physical access to the machines on which the application executes.

For instance, in a 2-tier Windows Forms or ASP.NET application, the machine running the interface code must have credentials to access the database server. Switching to a 3-tier model in which the data access code runs on an application server means that the machine running the interface code no longer needs those credentials, potentially making the system more secure.

Security requirements vary radically based on the environment and the requirements of your application. A Windows Forms application deployed only to internal users may need relatively little security, but an ASP.NET application exposed to anyone on the Internet may need extensive security.

To a large degree, security is all about surface area: how many points of attack are exposed from the application? The surface area can be defined in terms of domains of trust.

Security and Internal Applications Internal applications are totally encapsulated within a domain of trust: the client and all servers are running in a trusted environment. This means that virtually every part of the application is exposed to a potential hacker (assuming that the hacker can gain physical access to a machine on the network in the first place). In a typical organization, hackers can attack the client workstation, the web server, the application server, and the database server if they so choose. Rarely are there firewalls or other major security roadblocks *within* the context of an organization's local area network (LAN).

Note Obviously, there *is* security. It is common to use Windows domain or Active Directory security on the clients and servers, but there's nothing stopping someone from attempting to communicate directly with any of these machines. Within a typical LAN, users can usually connect through the network to all machines because of a lack of firewall or physical barriers.

Many internal applications are coming under increasing security requirements because of government regulations and other business pressures. The idea of having the database credentials on a client workstation is rapidly becoming unacceptable, and this is driving organizations to adopt a 3-tier architecture simply to move those credentials to a separate application server. This is an easy way to quickly improve an application's security.

Of course, the result is that the clients have the credentials to the *application* server. If they know how to find and call the application server's services, they can use an application's own services to access its servers in invalid ways. This problem was particularly acute with DCOM, because there were browsers that end users could use to locate and invoke server-side services. Thanks to COM, users could use Microsoft Excel to locate and interact with server-side COM components, thereby bypassing the portions of the application that were *supposed* to run on the client. This meant that the applications were vulnerable to power users who could use server-side components in ways their designers never imagined.

This problem is rapidly transferring to XML-based services, as Microsoft Office and other end-user applications start to allow power users to call XML-based services from within macros. I expect to find power users calling XML-based services in unexpected ways in the very near future.

The services in this book are designed to prevent casual usage of the objects, even if a power user were to gain access to the service from his application.

In summary, security has replaced scalability as the primary driver for moving from 2- to 3-tier architectures. But you must be careful when designing your services to ensure you haven't simply shifted the problem down a level.

Security and External Applications For external applications, things are entirely different. This is really where SOA comes into play. Service orientation (SO) is all about assembling a system that spans trust boundaries. When part of your system is deployed outside your own network, it certainly crosses at least a security (trust) boundary.

In a client/server model, this would be viewed as a minimum of two tiers, since the client workstation is physically separate from any machines running behind the firewall. But really, SO offers a better way to look at the problem: there are two totally separate applications. The client runs one application, and another application runs on your server. These two applications communicate with each other through clearly defined messages, and neither application is privy to the internal implementation of the other.

This provides a good way to deal with not only the security trust boundary but also with the *semantic* trust boundary. What I mean by this is that the server application assumes that any data coming from the client application is flawed: either maliciously or because of a bug or oversight in the client. Even if the client has *security* access to interact with your server, the server application cannot assume that the semantic meaning of the data coming from the client is valid.

In short, because the client workstations are outside the domain of trust, you should assume that they're compromised and potentially malicious. You should assume that any code running on those clients will run incorrectly or not at all; in other words, the client input must be completely validated as it enters the domain of trust, even if the client includes code to do the validation.

Note I've had people tell me that this is an overly paranoid attitude, but I've been burned too many times: any time an interface is exposed (Windows, web, XML, and so on) so that clients outside your control can use it, you should assume that the interface will be misused. Often, this misuse is unintentional—for example, someone may write a buggy macro to automate data entry. That's no different than if they made a typo while entering the data by hand, but user-entered data is always validated before being accepted by an application. The same must be true for automated data entry as well, or your application will fail.

This scenario occurs in three main architectures: Windows smart clients, rich Internet applications (RIAs), and SOA systems.

If you deploy a WPF or Windows Forms client application to external workstations, you should design it as a stand-alone application that calls your server application through services. Chapter 21 shows how you can do this with the object-oriented concepts in this book.

You may create an RIA with asynchronous JavaScript and XML (Ajax) technologies or newer technologies such as Silverlight. In either case, the RIA often validates data or otherwise provides a richer experience for the user, but your server code should assume that the RIA didn't do anything it was supposed to do. It is far too easy for a user to subvert your client-side JavaScript or otherwise bypass client-side processing—as such, nothing running in an RIA can be trusted. The code running in the browser should be viewed as a *separate* application that is not trusted by the server application.

Service-oriented systems imply that there's one or more (potentially unknown) applications out there consuming your services. The very nature of SOA means that you have no control over those applications, so it would be foolish to assume they'll provide valid input to your services. A healthy dose of paranoia is critical when building any service for an SOA system.

As you'll see, you can use the object-oriented concepts and techniques shown in this book to create smart client applications that call services on your servers. You can use the same concepts to create the services themselves. You can also use them to create web applications ranging from simple Web Forms to Ajax to Silverlight.

Fault Tolerance

You can achieve fault tolerance by identifying points of failure and providing redundancy. Typically, applications have numerous points of failure. Some of the most obvious are as follows:

- The network feed to your user's buildings
- The power feed to your user's buildings
- The network feed and power feed to your data center
- The primary DNS host servicing your domain
- Your firewall, routers, switches, and so on
- Your web server
- Your application server
- Your database server
- Your internal LAN

In order to achieve high levels of fault tolerance, you need to ensure that if any one of these fails, some system will instantly kick in and fill the void. If the data center power goes out, a generator will kick in. If a bulldozer cuts your network feed, you'll need to have a second network feed coming in from the other side of the building, and so forth.

Considering some of the larger and more well-known outages of major websites in the past couple of years, it's worth noting that most of them occurred because of construction work cutting network or power feeds or because their ISP or external DNS provider went down or was attacked. That said, there are plenty of examples of websites going down because of local equipment failure. The reason why the high-profile failures are seldom due to this type of problem is because large sites make sure to provide redundancy in these areas.

Clearly, adding redundant power, network, ISP, DNS, or LAN hardware will have little impact on application architecture. Adding redundant servers, on the other hand, *will* affect the n-tier application architecture—or at least the application design. Each time you add a physical tier, you need to ensure that you add redundancy to the servers in that tier. Thus, adding a fault-tolerant physical tier always means adding at least *two* servers to the infrastructure.

The more physical tiers, the more redundant servers there are to configure and maintain. This is why fault tolerance is typically expensive to achieve.

Not only that, but to achieve fault tolerance through redundancy, all servers in a tier must also be logically identical at all times. For example, at no time can a user be tied to a specific server, so no single server can ever maintain any user-specific information. As soon as a user is tied to a specific server, that server becomes a point of failure for that user. The result is that the user loses fault tolerance.

Achieving a high degree of fault tolerance isn't easy. It requires a great deal of thought and effort to locate all points of failure and make them redundant. Having fewer physical tiers in an architecture can assist in this process by reducing the number of tiers that must be made redundant.

To summarize, the number of physical tiers in an architecture is a trade-off between performance, scalability, security, and fault tolerance. Furthermore, the optimal configuration for a web application isn't the same as the one for an intranet application with smart client machines. If an application framework is to have any hope of broad appeal, it needs flexibility in the physical architecture so that it can support web and smart clients effectively, as well as provide both with optimal performance and scalability. Beyond that, it needs to work well in a service-oriented environment to create both client and server applications that interact through message-based communication.

A 5-Layer Logical Architecture

This book will explore a 5-layer logical architecture and show how you can implement it using object-oriented concepts. Once you learn how to create the logical architecture, you'll discover how to configure it into various physical architectures in order to achieve optimal results for WPF, Windows Forms, ASP.NET, and service-oriented and workflow interfaces.

Note If you get any group of architects into a room and ask them to describe their ideal architecture, each one will come up with a different answer. I make no pretense that this architecture is the only one out there, nor do I intend to discuss all the possible options. My aim here is to present a coherent, distributed, object-oriented architecture that supports all these different interfaces.

In the framework used in this book, the logical architecture comprises the five layers shown in Figure 1-1.

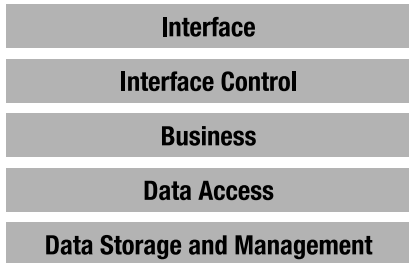


Figure 1-1. *The 5-layer logical architecture*

Remember that the benefit of a logical n-layer architecture is the separation of functionality into clearly defined roles or groups in order to increase clarity and maintainability. Let's define each of the layers more carefully.

Interface

The Interface layer is often referred to as the UI or Presentation layer. I am using the more generic term *Interface*, because this architecture supports service-oriented applications that have no user, as well as WPF, Web Forms, Windows Forms, and other application types that *do* have a user.

At first, it may not be clear why I've separated Interface from Interface Control. Certainly, from a smart client perspective, the interface and the control of the interface are one and the same: they are graphical user interface (GUI) forms with which the user can interact.

From a web perspective, the distinction is probably quite clear. Typically, the browser merely provides an interface for the user, displaying data and collecting user input. In that case, all the actual interaction logic—the code written to control the interface, to *generate* the output, or to *interpret* user input—runs on the web server (or mainframe) and not on the client machine.

Of course, in today's world, the browser might run Ajax or Silverlight. But as discussed earlier in the chapter, none of this code can be trusted. It must be viewed as being a *separate* application that interacts with your application as it runs on the server. So even with code running in the browser, *your* application's interface code is running on your web server.

The same is true for an SOA system, where the consuming application is clearly separate and thus can't be trusted. Your application's interface is composed of XML messages, and your interface control code (the service implementation) is running on your server.

Knowing that the logical model must support both smart and web-based clients (along with even more limited clients, such as cell phones or other mobile devices), it's important to recognize that in many cases, the interface will be physically separate from the interface control logic. To accommodate this separation, you need to design the applications around this concept.

Note The types of interface technologies continue to multiply, and each comes with a new and relatively incompatible technology with which you must work. It's virtually impossible to create a programming framework that entirely abstracts interface concepts. Because of this, the architecture and framework will merely *support the creation* of varied interfaces, not automate their creation. Instead, the focus will be on simplifying the other layers in the architecture, for which technology is more stable.

Interface Control

Now that I've addressed the distinction between the Interface and the Interface Control layers, the latter's purpose is probably fairly clear. This layer includes the logic to decide what the user sees, the

navigation paths, and how to interpret user input. In a WPF or Windows Forms application, this is the code behind the form. Actually, it's the code behind the form in a Web Forms application, too, but here it can also include code that resides in server-side controls; *logically*, that's part of the same layer.

In an ASP.NET MVC application, the view and controller are both part of the Interface Control layer. The HTML, JavaScript, and other content produced by the view comprise the Interface. Finally, the Business layer is the model.

In many applications, the interface control code is very complex. For starters, it must respond to the user's requests in a nonlinear fashion. (It is difficult to control how users might click controls or enter or leave the forms or pages.) The interface control code must also interact with logic in the Business layer to validate user input, to perform any processing that's required, or to do any other business-related action.

Basically, the goal is to write interface control code that accepts user input and then provides it to the Business layer, where it can be validated, processed, or otherwise manipulated. The interface control code must then respond to the user by displaying the results of its interaction with the Business layer. Was the user's data valid? If not, what was wrong with it? And so forth.

In .NET, the interface control code is almost always event-driven. WPF and Windows Forms code is all about responding to events as the user types and clicks the form, and ASP.NET code is all about responding to events as the browser round-trips the user's actions back to the web server. Although WPF, Windows Forms, and ASP.NET technologies make heavy use of objects, the code that is typically written for the Interface Control layer isn't object-oriented as much as procedural and event-based.

That said, there's great value in creating frameworks and reusable components that support a particular type of interface. When creating a WPF or Windows Forms interface, developers can use numerous object-oriented techniques to simplify the creation, display, and management of the forms. When creating a web interface, developers can use ASP.NET master pages, user controls, and custom server controls to provide reusable components that simplify page development.

Because there's such a wide variety of interface styles and approaches, I won't spend much time dealing with interface development or frameworks in this book. Instead, I'll focus on simplifying the creation of the business logic and data access layers, which are required for any type of interface.

Business Logic

Business logic includes all business rules, data validation, manipulation, processing, and authorization for the application. One definition from Microsoft, which has since been taken down from MSDN, is as follows: "The combination of validation edits, login verifications, database lookups, policies, and algorithmic transformations that constitute an enterprise's way of doing business."

Note Again, although you may implement validation logic to run in a browser or other external client, you can't trust that code. You must view the logic that runs under your control in the Business layer as being the only *real* validation logic.

The business logic *must* reside in a separate layer from the interface code. Although you may choose to duplicate some of this logic in your interface control code to provide a richer user experience, the Business layer must implement all the business logic, because it is the only point of central control and maintainability.

I believe that this particular separation between the responsibilities of the Business layer and Interface Control layer is absolutely critical if you want to gain the benefits of increased maintainability and reusability. This is because any business logic that creeps into the Interface or Interface

Control layer will reside within a *specific* interface and will not be available to any other interfaces that you might create later.

Any business logic written into, say, a WPF interface is useless to a web or service interface and must therefore be written into those as well. This instantly leads to duplicated code, which is a maintenance nightmare. You can separate these two layers through techniques such as clearly defined procedural models or object-oriented design and programming. In this book, I'll show how to use object-oriented concepts to help separate the business logic from the interface.

It is important to recognize that a typical application will use business logic in a couple of different ways. Most applications have some user interaction, such as forms in which the user views or enters data into the system. Most applications also have some very non-interactive processes, such as posting invoices, relieving inventory, or calculating insurance rates.

Ideally, the Business layer will be used in a very rich and interactive way when the user is entering data directly into the application. For instance, when a user is entering a sales order, she expects that the validation of data, the calculation of tax, and the subtotaling of the order will happen literally as she types. This implies that the Business layer can be physically deployed on the client workstation or on the web server to provide the high levels of interactivity users desire.

To support non-interactive processes, on the other hand, the Business layer often needs to be deployed onto an application server, or as close to the database server as possible. For instance, the calculation of an insurance rate can involve extensive database lookups along with quite a bit of complex business processing. This is the kind of thing that should occur behind the scenes on a server, not on a user's desktop.

Fortunately, it is possible to deploy a logical layer on multiple physical tiers. Doing this does require some up-front planning and technical design, as you'll see in Chapter 2. The end result, however, is a single Business layer that is potentially deployed both on the client workstation (or web server) and on the application server. This allows the application to provide high levels of interactivity when the user is working directly with the application and to provide efficient back-end processing for non-interactive processes.

Data Access

Data access code interacts with the Data Storage and Management layer to retrieve, insert, update, and remove information. The Data Access layer doesn't actually manage or store the data; it merely provides an interface between the business logic and the database.

Data access gets its own logical layer for much the same reason that the interface is split from interface control. In some cases, data access will occur on a machine that's physically separate from the one on which the interface and/or business logic is running. In other cases, data access code will run on the same machine as the business logic (or even the interface) in order to improve performance or fault tolerance.

Note It may sound odd to say that putting the Data Access layer on the same machine as the business logic can *increase* fault tolerance, but consider the case of web farms, in which each web server is identical to all the others. Putting the data access code on the web servers provides automatic redundancy of the Data Access layer along with the Business and Interface layers.

Adding an extra physical tier just to do the data access makes fault tolerance harder to implement, because it increases the number of tiers in which redundancy needs to be implemented. As a side effect, adding more physical tiers also reduces performance for a single user, so it's not something that should be done lightly.

Logically defining data access as a separate layer enforces a separation between the business logic and any interaction with a database (or any other data source). This separation provides the flexibility to choose later whether to run the data access code on the same machine as the business logic or on a separate machine. It also makes it much easier to change data sources without affecting the application. This is important because it enables switching from one database vendor to another at some point.

This separation is useful for another reason: Microsoft has a habit of changing data access technologies every three years or so, meaning that it is necessary to rewrite the data access code to keep up (remember DAO, RDO, ADO 1.0, ADO 2.0, ADO.NET, and now LINQ and the ADO.NET Entity Framework?). By isolating the data access code into a specific layer, the impact of these changes is limited to a smaller part of the application.

Data access mechanisms are typically implemented as a set of services; each service is a procedure that the business logic calls to retrieve, insert, update, or delete data. Although these services are often constructed using objects, it's important to recognize that the designs for an effective Data Access layer are really quite procedural in nature. Attempts to force more object-oriented designs for relational database access often result in increased complexity or decreased performance. I think the best approach is to implement the data access as a set of methods but encapsulate those methods within objects to keep them organized logically.

Note If you're using an object database instead of a relational database, then of course the data access code may be very object-oriented. Few of us get such an opportunity, however, because almost all data is stored in relational databases.

Sometimes the Data Access layer can be as simple as a series of methods that use ADO.NET directly to retrieve or store data. In other circumstances, the Data Access layer is more complex, providing a more abstract or even metadata-driven way to get at data. In these cases, the Data Access layer can contain a lot of complex code to provide this more abstract data access scheme. The framework created in this book doesn't restrict how you implement your Data Access layer. The examples in the book use LINQ to SQL, but you could also use ADO.NET or use some other metadata-driven Data Access layer directly if you prefer.

Another common role for the Data Access layer is to provide mapping between the object-oriented business logic and the relational data in a data store. A good object-oriented model is almost never the same as a good relational database model. Objects often contain data from multiple tables, or even from multiple databases; or conversely, multiple objects in the model can represent a single table. The process of taking the data from the tables in a relational model and getting it into the object-oriented model is called *object-relational mapping* (ORM), and I'll have more to say on the subject in Chapter 2.

Data Storage and Management

Finally, there's the Data Storage and Management layer. Database servers such as SQL Server and Oracle often handle these tasks, but, increasingly, other applications may provide this functionality, too, via technologies such as XML-based services.

What's key about this layer is that it handles the physical creation, retrieval, update, and deletion of data. This is different from the Data Access layer, which *requests* the creation, retrieval, update, and deletion of data. The Data Storage and Management layer actually *implements* these operations within the context of a database or a set of files, and so on.

The business logic (via the Data Access layer) invokes the Data Storage and Management layer, but the layer often includes additional logic to validate the data and its relationship to other data.

Sometimes, this is true relational data modeling from a database; other times, it's the application of business logic from an external application. What this means is that a typical Data Storage and Management layer will include business logic that is also implemented in the Business layer. This time the replication is unavoidable because relational databases are designed to enforce data integrity, and that's just another form of business logic.

In summary, whether you're using stored procedures in SQL Server or service calls to another application, you typically handle data storage and management by creating a set of services or procedures that you can call as needed. Like the Data Access layer, it's important to recognize that the designs for data storage and management are typically very procedural.

Table 1-1 summarizes the five layers and their roles.

Table 1-1. *The Five Logical Layers and the Roles They Provide*

Layer	Roles
Interface	Renders display and collects user input.
Interface Control	Acts as an intermediary between the user and the business logic, taking user input and providing it to the business logic, then returning results to the user.
Business Logic	Provides all business rules, validation, manipulation, processing, and security for the application.
Data Access	Acts as an intermediary between the business logic and data management. Also encapsulates and contains all knowledge of data access technologies (such as LINQ to SQL), databases, and data structures.
Data Storage and Management	Physically creates, retrieves, updates, and deletes data in a persistent data store.

Everything I've talked about to this point is part of a *logical* architecture. Now it's time to move on and see how you can apply it in various *physical* configurations.

Applying the Logical Architecture

Given this 5-layer logical architecture, it should be possible to configure it into one, two, three, four, or five physical tiers in order to gain performance, scalability, security, or fault tolerance to various degrees and in various combinations.

Note In this discussion, I assume that you have total flexibility to configure which logical layer runs where. In some cases, technical issues may prevent the physical separation of some layers. As I noted earlier, you need to strike a balance between flexibility and capability.

I want to discuss a few physical configurations to illustrate how the logical model works. These are common and important setups that are encountered on a day-to-day basis.

Optimal Performance Smart Client

When so much focus is placed on distributed systems, it's easy to forget the value of a single-tier solution. Point of sale, sales force automation, and many other types of applications often run in

stand-alone environments. However, the benefits of the logical n-layer architecture are still desirable in terms of maintainability and code reuse.

It probably goes without saying that everything can be installed on a single client workstation. An optimal performance smart client is usually implemented using WPF or Windows Forms for the interface, with the business logic and data access code running in the same process and talking to a Microsoft SQL Server Express or a Microsoft SQL Server Compact Edition database. The fact that the system is deployed on a single physical tier doesn't compromise the logical architecture and separation, as shown in Figure 1-2.

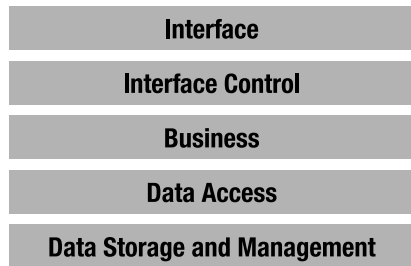


Figure 1-2. *The five logical layers running on a single machine*

It's important to remember that n-layer systems can run on a single machine in order to support the wide range of applications that require stand-alone machines. It's also worth pointing out that this is basically the same as a 2-tier, "fat-client" physical architecture; the only difference is that in a 2-tier physical architecture, the Data Storage and Management layer would be running on a central database server, such as SQL Server or Oracle, as shown in Figure 1-3.

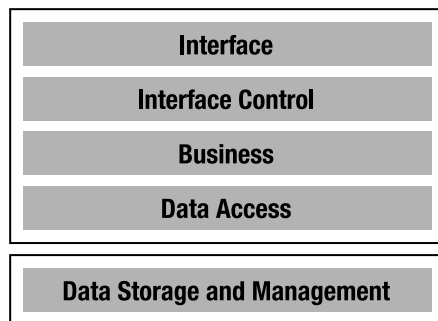


Figure 1-3. *The five logical layers with a separate database server*

Other than the location of the data storage, this is identical to the single-tier configuration, and typically the switch from single-tier to 2-tier revolves around little more than changing the database connection string.

High-Scalability Smart Client

Single-tier configurations are good for stand-alone environments, but they don't scale well. To support multiple users, it is common to use 2-tier configurations. I've seen 2-tier configurations support more than 350 concurrent users against SQL Server with very acceptable performance.

Going further, it is possible to trade performance to gain scalability by moving the Data Access layer to a separate machine. Single-tier or 2-tier configurations give the best performance, but they don't scale as well as a 3-tier configuration would. A good rule of thumb is that if you have more than 50 to 100 concurrent users, you can benefit by using a separate server to handle the Data Access layer.

Another reason for moving the Data Access layer to an application server is security. Since the Data Access layer contains the code that interacts directly with the database, the machine on which it runs must have credentials to access the database server. Rather than having those credentials on the client workstation, they can be moved to an application server. This way, the user's computer won't have the credentials to interact directly with the database server, thus increasing security.

It is also possible to put the Business layer on the application server. This is useful for non-interactive processes such as batch updates or data-intensive business algorithms. Yet, at the same time, most applications allow for user interaction, so there is a definite need to have the Business layer running on the client workstation to provide high levels of interactivity for the user.

As discussed earlier in the chapter, it is possible to deploy the same logical layer onto multiple physical tiers. Using this idea, you can put the Data Access layer on an application server, and you can put the Business layer on *both* the client workstation and the application server, as shown in Figure 1-4.

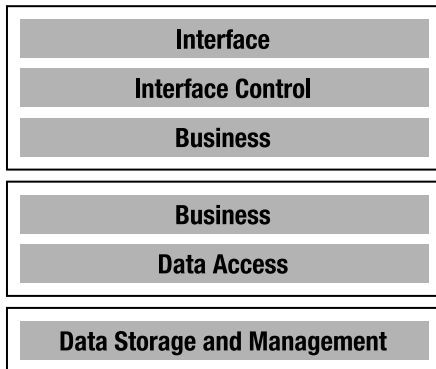


Figure 1-4. *The five logical layers with separate application and database servers*

Putting the Data Access layer on the application server centralizes all access to the database on a single machine. In .NET, if the connections to the database for all users are made using the same user ID and password, you'll get the benefits of *connection pooling* for all your users. What this means immediately is that there will be far fewer connections to the database than there would have been if each client machine had connected directly. The actual reduction depends on the specific application, but often it means supporting 150 to 200 concurrent users with just two or three database connections.

Of course, all user requests now go across an extra network hop, thereby causing increased latency (and therefore decreased performance). This performance cost translates into a huge scalability gain, however, because this architecture can handle many more concurrent users than a 2-tier physical configuration.

With the Business layer deployed on both the client and server, the application is able to fully exploit the strengths of both machines. Validation and a lot of other business processing can run on the client workstation to provide a rich and highly interactive experience for the user, while non-interactive processes can efficiently run on the application server.

If well designed, such an architecture can support *thousands* of concurrent users with adequate performance.

Optimal Performance Web Client

As with WPF or Windows Forms applications, the best performance is received in web-based applications by minimizing the number of physical tiers. However, the trade-off in a web scenario is different: in this case, it is possible to improve performance and scalability at the same time but at the cost of security, as I will demonstrate.

To get optimal performance in a web application, it is desirable to run most of the code in a single process on a single machine, as shown in Figure 1-5.

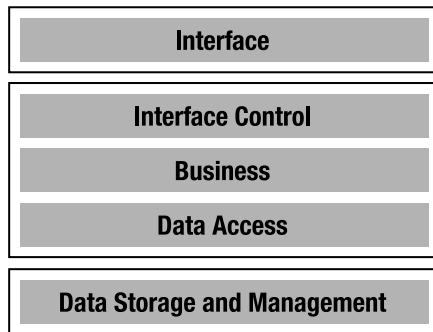


Figure 1-5. *The five logical layers as used for web applications*

The Interface layer must be physically separate because it's running in a browser, but the Interface Control, Business, and Data Access layers can all run on the same machine, in the same process. In some cases, you might even put the Data Storage and Management layer on the same physical machine, though this is suitable only for smaller applications.

This minimizes network and communication overhead and optimizes performance. Figure 1-6 shows how it is possible to get good scalability, because the web server can be part of a web farm in which all the web servers are running the same code.

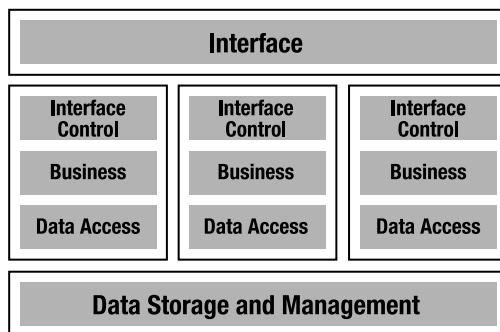


Figure 1-6. *The five logical layers deployed on a load-balanced web farm*

This setup provides good database-connection pooling, because each web server will be (potentially) servicing hundreds of concurrent users, and all database connections on a web server are pooled.

Unless the database server is getting overwhelmed with connections from the web servers in the web farm, a separate application server will rarely provide gains in scalability. If a separate application server is needed, there will be a reduction in performance because of the additional physical tier. (Let's hope there will be a gain in scalability, because the application server can consolidate database connections across all the web servers.) It is important to consider fault tolerance in this case, because redundant application servers may be needed in order to avoid a point of failure.

Another reason for implementing an application server is to increase security, and that's the topic of the next section.

High-Security Web Client

As discussed in the earlier section on security, you'll have many projects that dictate that a web server can never talk directly to a database. The web server must run in a demilitarized zone (DMZ), sandwiched between the external firewall and a second internal firewall. The web server must communicate with another server through the internal firewall in order to interact with the database or any other internal systems.

As with the 3-tier Windows client scenario, there is tremendous benefit to also having the Business layer deployed on both the web server and the application server. Such a deployment allows the web interface control code to interact closely with the business logic when appropriate, while non-interactive processes can simply run on the application server. This deployment is illustrated in Figure 1-7, in which the dashed lines represent the firewalls.

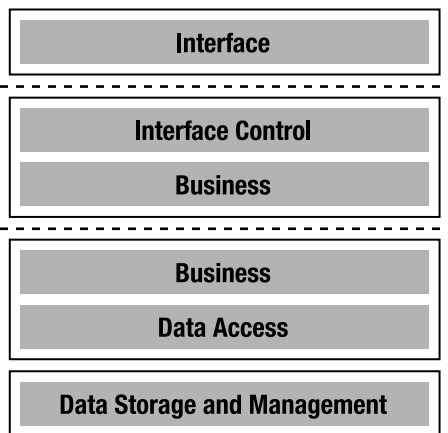


Figure 1-7. *The five logical layers deployed in a secure web configuration*

Splitting out the Data Access layer and running it on a separate application server increases the security of the application. However, this comes at the cost of performance—as discussed earlier, this configuration will typically cause a performance degradation of around 50 percent. Scalability, on the other hand, is fine: like the first web configuration, you can achieve scalability by implementing a web farm in which each web server runs the same interface control and business logic code, as shown in Figure 1-8.

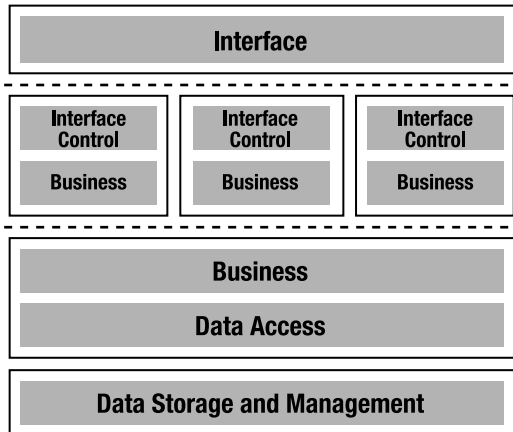


Figure 1-8. *The five logical layers in a secured environment with a web farm*

The Way Ahead

After implementing the framework to support this 5-layer architecture, I'll create a sample application with three different interfaces: WPF, web, and XML-based Windows Communication Foundation (WCF) services. This will give you the opportunity to see firsthand how the framework supports the following models:

- High-scalability smart client
- Optimal performance web client
- Optimal performance web service

Because of the way the framework is implemented, switching to any of the other models just discussed will require only configuration file changes. The result is that you can easily adapt your application to any of the physical configurations without having to change your code.

Managing Business Logic

At this point, you should have a good understanding of logical and physical architectures and how you can configure a 5-layer logical architecture into various n-tier physical architectures. In one way or another, all of these layers will use or interact with the application's data. That's obviously the case for the Data Storage and Management and Data Access layers, but the Business layer must validate, calculate, and manipulate data; the Interface Control layer transfers data between the Business and Interface layers (often performing formatting or using the data to make navigational choices); and the Interface layer displays data to the user and collects new data as it's entered.

In an ideal world, all of the business logic would exist in the Business layer, but in reality, this is virtually impossible to achieve. In a web-based application, validation logic is often included in the Interface layer, so that the user gets a more interactive experience in the browser. Unfortunately, any validation that's done in the web browser is unreliable, because it's too easy for a malicious user to bypass that validation. Thus, any validation done in the browser must be rechecked in the Business layer as well.

Similarly, most databases enforce data integrity and often some other rules, too. Furthermore, the Data Access layer often includes business logic to decide when and how data should be stored or retrieved from databases and other data sources. In almost any application, to a greater or a lesser extent, business logic gets scattered across all the layers.

There's one key truth here that's important: for each piece of application data, there's a fixed set of business logic associated with that data. If the application is to function properly, the business logic must be applied to that data at least once. Why "at least"? Well, in most applications, some of the business logic is applied more than once. For example, a validation rule applied in the Interface layer can be reapplied in the Interface Control layer or Business layer before data is sent to the database for storage. In some cases, the database includes code to recheck the value as well.

Now, I'd like to cover some of the more common options. I'll start with three popular (but flawed) approaches. Then I'll discuss a compromise solution that's enabled through the use of mobile objects, such as the ones supported by the framework I'll create later in the book.

Potential Business Logic Locations

Figure 1-9 illustrates common locations for validation and manipulation business logic in a typical application. Most applications have the same logic in at least a couple of these locations.

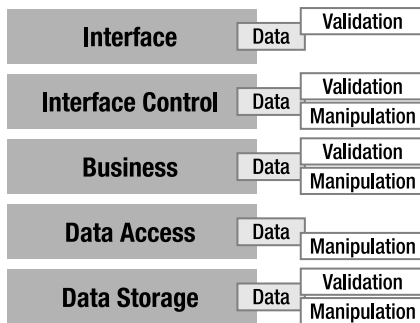


Figure 1-9. Common locations for business logic in applications

Business logic is put in a web Interface layer to give the user a more interactive experience—and put into a Windows Interface layer for the same reason. The business logic is rechecked on the web server because the browser isn't trustworthy. And database administrators put the logic into the database (via stored procedures and other database constructs) because they don't trust any application developers.

The result of all this validation is a lot of duplicated code, all of which has to be debugged, maintained, and somehow kept in sync as the business needs (and thus logic) change over time. In the real world, the logic is almost never *really* kept in sync, so developers must constantly debug and maintain the code in a near-futile effort to make all these redundant bits of logic agree with each other.

One solution is to force all the logic into a single layer, thereby making the other layers as "dumb" as possible. There are various approaches to this, although (as you'll see) none of them provides an optimal solution.

Another solution is to dynamically generate the validation logic for the Interface or Interface Control layer based on metadata provided from the Business layer. This requires more work in the interface layers but can increase maintainability overall.

Business Logic in the Data Storage and Management Layer

The classic approach is to put all logic into the database as the single, central repository. The interface then allows the user to enter absolutely anything (because any validation would be redundant), and the Business layer now resides inside the database. The Data Access layer does nothing but move the data into and out of the database, as shown in Figure 1-10.

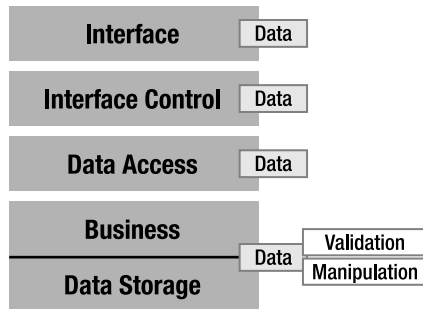


Figure 1-10. Validation and business logic in the Data Storage and Management layer

The advantage of this approach is that the logic is centralized, but the drawbacks are plentiful. For starters, the user experience is totally non-interactive. Users can't get any results, or even confirmation that their data is valid, without round-tripping the data to the database for processing. The database server becomes a performance bottleneck, because it's the only thing doing any actual work. Unfortunately, the hardest physical tier to scale up for more users is the database server, since it is difficult to use load-balancing techniques on it. The only real alternative is to buy bigger and bigger server machines.

Business Logic in the Interface Control Layer

Another common approach is to put all the business logic into the interface control code. The data is validated and manipulated by the interface control code, and the Data Storage and Management layer just stores the data. This approach, as shown in Figure 1-11, is common in both Windows and web environments and has the advantage that the business logic is centralized into a single tier (and of course, one can write the business logic in a language such as C# or VB .NET).

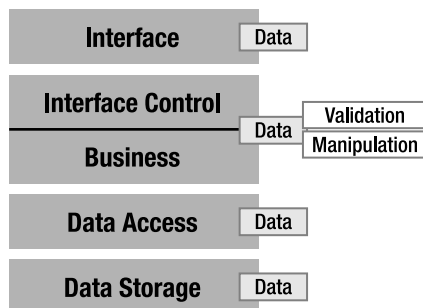


Figure 1-11. Business logic deployed with only the Interface Control layer

Unfortunately, in practice, the business logic ends up being scattered throughout the application, thereby decreasing readability and making maintenance more difficult. Even more important, business logic in one form or page isn't reusable when subsequent forms or pages are created that use the same data. Furthermore, in a web environment, this architecture also leads to a totally non-interactive user experience, because no validation can occur in the browser. The user must transmit his data to the web server for any validation or manipulation to take place.

Note ASP.NET validation controls at least allow for basic data validation in the interface, with that validation automatically extended to the browser by the ASP.NET technology itself. Though not a total solution, this is a powerful feature that does help.

Business Logic in the Middle (Merged Business and Data Access Layers)

Still another option is the classic UNIX client/server approach, whereby the Business and Data Access layers are merged, keeping the Interface, Interface Control, and Data Storage and Management layers as “dumb” as possible (see Figure 1-12).

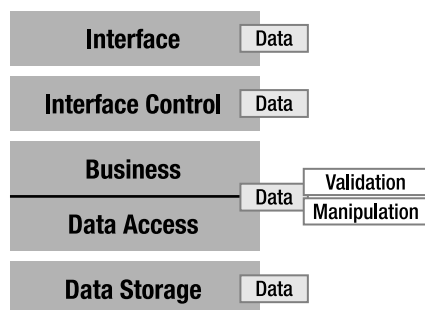


Figure 1-12. *Business logic deployed only on the application server*

Unfortunately, once again, this approach falls afoul of the non-interactive user experience problem: the data must round-trip to the Business/Data Access layer for any validation or manipulation. This is especially problematic if the Business/Data Access layer is running on a separate application server, because then you're faced with network latency and contention issues, too. Also, the central application server can become a performance bottleneck, because it's the only machine doing any work for all the users of the application.

Sharing Business Logic Across Tiers

I wish this book included the secret that allows you to write all your logic in one central location, thereby avoiding all of these awkward issues. Unfortunately, that's not possible with today's technology: putting the business logic only on the client, application server, or database server is problematic, for all the reasons given earlier. But something needs to be done about it, so what's left?

What's left is the possibility of centralizing the business logic in a Business layer that's deployed on the client (or web server), so that it's accessible to the Interface Control layer, and in a Business layer that's deployed on the application server, so that it's able to interact efficiently with the Data Access layer. The end result is the best of both worlds: a rich and interactive user experience and efficient high-performance back-end processing when interacting with the database (or other data source).

In the simple cases in which there is no application server, the Business layer is deployed only once: on the client workstation or web server, as shown in Figure 1-13.

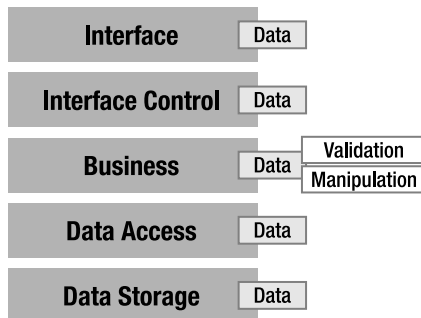


Figure 1-13. Business logic centralized in the Business layer

Ideally, this business logic will run on the same machine as the interface control code when interacting with the user, but on the same machine as the data access code when interacting with the database. (As discussed earlier, all of this could be on one machine or a number of different machines, depending on your physical architecture.) It must provide a friendly interface that the interface developer can use to invoke any validation and manipulation logic, and it must also work efficiently with the Data Access layer to get data in and out of storage.

The tools for addressing this seemingly intractable set of requirements are *mobile business objects* that encapsulate the application's data along with its related business logic. It turns out that a properly constructed business object can move around the network from machine to machine with almost no effort on your part. The .NET Framework itself handles the details, and you can focus on the business logic and data.

By properly designing and implementing mobile business objects, you allow the .NET Framework to pass your objects across the network *by value*, thereby copying them automatically from one machine to another. This means that with little extra code, you can have your business logic and business data move to the machine where the Interface Control layer is running and then shift to the machine where the Data Access layer is running when data access is required.

At the same time, if you're running the Interface Control and Data Access layers on the same machine, then the .NET Framework won't move or copy your business objects. They're used directly by both tiers with no performance cost or extra overhead. You don't have to do anything to make this happen, either—.NET automatically detects that the object doesn't need to be copied or moved and thus takes no extra action.

The Business layer becomes portable, flexible, and mobile, and it adapts to the physical environment in which you deploy the application. Because of this, you're able to support a variety of physical n-tier architectures with one code base, whereby your business objects contain no extra code to support the various possible deployment scenarios. What little code you need to implement to support the movement of your objects from machine to machine will be encapsulated in a framework, leaving the business developer to focus purely on the development of business logic.

Business Objects

Having decided to use business objects and take advantage of .NET's ability to move objects around the network automatically, it's now time to discuss business objects in more detail. I will discuss exactly what they are and how they can help you centralize the business logic pertaining to your data.

The primary goal when designing any kind of software object is to create an abstract representation of some entity or concept. In ADO.NET, for example, a `DataTable` object represents a tabular set of data. `DataTable` objects provide an abstract and consistent mechanism by which you can work with *any* tabular data. Likewise, a Windows Forms `TextBox` control is an object that represents the concept of displaying and entering data. From the application's perspective, there is no need to have any understanding of how the control is rendered on the screen or how the user interacts with it. It's just an object that includes a `Text` property and a handful of interesting events.

Key to successful object design is the concept of *encapsulation*. This means that an object is a black box: it contains logic and data, but the user of the object doesn't know *what* data or *how* the logic actually works. All the user can do is interact with the object.

Note Properly designed objects encapsulate both behavior or logic and the data required by that logic.

If objects are abstract representations of entities or concepts that encapsulate both data and its related logic, what then are *business objects*?

Note Business objects are different from regular objects only in terms of what they represent.

Object-oriented applications are created to address problems of one sort or another. In the course of doing so, a variety of different objects are often used. Some of these objects will have no direct connection with the problem at hand (`DataTable` and `TextBox` objects, for example, are just abstract representations of computer concepts). However, others will be closely related to the area or *domain* in which you're working. If the objects are related to the business for which you're developing an application, then they're business objects.

For instance, if you're creating an order entry system, your business domain will include things such as customers, orders, and products. Each of these will likely become business objects within your order entry application—the `Order` object, for example, will provide an abstract representation of the order being placed by a customer.

Note Business objects provide an abstract representation of entities or concepts that are part of the business or problem domain.

Business Objects As Smart Data

I've already discussed the drawbacks of putting business logic into the Interface Control layer, but I haven't thoroughly discussed the drawback of keeping the data in a generic representation such as a `DataSet` or data transfer object (DTO). The data in a `DataSet` (or a DTO, array, or XML document) is unintelligent, unprotected, and generally unsafe. There's nothing to prevent anyone from putting invalid data into any of these containers, and there's nothing to ensure that the business logic behind one form in the application will interact with the data in the same way as the business logic behind another form.

A `DataSet` or an XML document with an XML Schema Definition (XSD) might ensure that text cannot be entered where a number is required or that a number cannot be entered where a date is required. At best, it might enforce some basic relational-integrity rules. However, there's no way to ensure that the values match other criteria, or that calculations or other processing is done properly

against the data, without involving other objects. The data in a `DataSet`, array, or XML document isn't self-aware; it's not able to apply business rules or handle business manipulation or processing of the data.

The data in a business object, however, is what I like to call *smart data*. Not only does the object contain the data, but it also includes all the business logic that goes along with that data. Any attempt to work with the data must go through this business logic. In this arrangement, there is much greater assurance that business rules, manipulation, calculations, and other processing will be executed consistently everywhere in the application. In a sense, the data has become self-aware and can protect itself against incorrect usage.

In the end, an object doesn't care whether it's used by a WPF interface, a batch-processing routine, or a web service. The code using the object can do as it pleases; the object itself will ensure that all business rules are obeyed at all times.

Contrast this with a `DataSet` or an XML document, in which the business logic doesn't reside in the data container but somewhere else—typically, a Windows or web form. If multiple forms or pages use this `DataSet`, there is no assurance that the business logic will be applied consistently. Even if you adopt a standard that says that interface developers must invoke methods from a centralized class to interact with the data, there's nothing preventing them from using the `DataSet` directly. This may happen accidentally or because it was simply easier or faster to use the `DataSet` than to go through some centralized routine.

Note With consistent use of business objects, there's no way to bypass the business logic. The only way to the data is through the object, and the object always enforces the rules.

So, a business object that represents an invoice will include not only the data pertaining to the invoice but also the logic to calculate taxes and amounts due. The object should understand how to post itself to a ledger and how to perform any other accounting tasks that are required. Rather than passing raw invoice data around and having the business logic scattered throughout the application, it is possible to pass an `Invoice` object around. The entire application can share not only the data but also its associated logic. Smart data through objects can dramatically increase the ability to reuse code and can decrease software maintenance costs.

Anatomy of a Business Object

Putting all of these pieces together, you get an object that has an interface (a set of properties and methods), some implementation code (the business logic behind those properties and methods), and state (the data). This is illustrated in Figure 1-14.

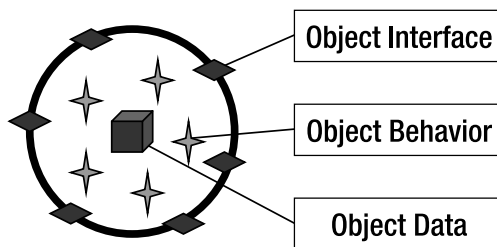


Figure 1-14. A business object composed of state, implementation, and interface

The hiding of the data and the implementation code behind the interface are keys to the successful creation of a business object. If the users of an object are allowed to “see inside” it, they will be tempted to cheat and to interact with the logic or data in unpredictable ways. This danger is the reason why it is important to take care when using the `Public` keyword as you build your classes.

Any property, method, event, or field marked as `Public` is available to the users of objects created from the class. For example, you might create a simple class such as the following:

```
Public Class Project
    Private _id As Guid = Guid.NewGuid()
    Public ReadOnly Property Id() As Guid
        Get
            Return _id
        End Get
    End Property
    Private _name As String = String.Empty
    Public Property Name() As String
        Get
            Return _name
        End Get
        Set(ByVal value As String)
            If value Is Nothing Then
                value = String.Empty
            End If
            If value.Length > 50 Then
                Throw New Exception("Name too long")
            End If
            _name = value
        End Set
    End Property
End Class
```

This defines a business object that represents a project of some sort. All that is known at the moment is that these projects have an ID value and a name. Notice, though, that the fields containing this data are `Private`—you don’t want the users of your object to be able to alter or access them directly. If they were `Public`, the values could be changed without the object’s knowledge or permission. (The `_name` field could be given a value that’s longer than the maximum of 50 characters, for example.)

The properties, on the other hand, are `Public`. They provide a controlled access point to the object. The `Id` property is read-only, so the users of the object can’t change it. The `Name` property allows its value to be changed but enforces a business rule by ensuring that the length of the new value doesn’t exceed 50 characters.

Note These concepts are not unique to business objects—they’re common to all objects and are central to object-oriented design and programming.

Mobile Objects

Unfortunately, directly applying the kind of object-oriented design and programming I’ve been talking about so far is often difficult in today’s complex computing environments. Object-oriented programs are almost always designed with the assumption that all the objects in an application can interact with each other with no performance penalty. This is true when all the objects are running in the same process on the same computer, but it’s not at all true when the objects might be running in different processes or even on different computers.

Earlier in this chapter, I discussed various physical architectures in which different parts of an application might run on different machines. With a high-scalability smart client architecture, for example, there will be a client, an application server, and a data server. With a high-security web client architecture, there will be a client, a web server, an application server, and a data server. Parts of the application will run on each of these machines, interacting with each other as needed.

In these distributed architectures, you can't use a straightforward object-oriented design, because any communication between classic fine-grained objects on one machine and similar objects on another machine will incur network latency and overhead. This translates into a performance problem that simply can't be ignored. To overcome this problem, most distributed applications haven't used object-oriented designs. Instead, they consist of a set of procedural code running on each machine, with the data kept in a *DataSet*, an array, or an XML document that's passed around from machine to machine.

This isn't to say that object-oriented design and programming are irrelevant in distributed environments—just that it becomes complicated. To minimize the complexity, most distributed applications are object-oriented *within a tier*, but between tiers they follow a procedural or service-based model. The end result is that the application as a whole is neither object-oriented nor procedural but a blend of both.

Perhaps the most common architecture for such applications is to have the Data Access layer retrieve the data from the database into a *DataSet*. The *DataSet* is then returned to the client (or the web server). The code in the forms or pages then interacts with the *DataSet* directly, as shown in Figure 1-15.

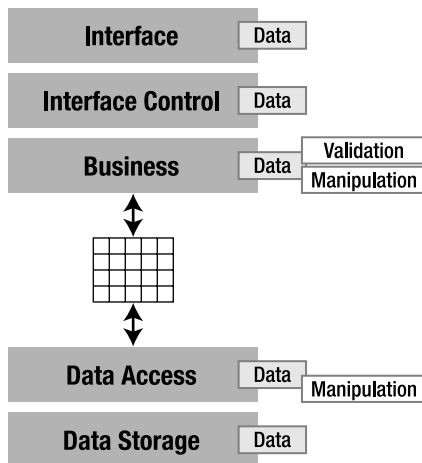


Figure 1-15. Passing a *DataSet* between the *Business* and *Data Access* layers

This approach has the maintenance and code-reuse flaws that I've talked about, but the fact is that it gives pretty good performance in most cases. Also, it doesn't hurt that most programmers are pretty familiar with the idea of writing code to manipulate a *DataSet*, so the techniques involved are well understood, thus speeding up development.

A decision to stick with an object-oriented approach should be undertaken carefully. It's all too easy to compromise the object-oriented design by taking the data out of the objects running on one machine, sending the raw data across the network, and allowing other objects to use that data outside the context of the objects and business logic. Such an approach would break the encapsulation provided by the logical Business layer.

Mobile objects are all about sending smart data (objects) from one machine to another, rather than sending raw data.

Through its WCF, serialization, and deployment technologies, the .NET Framework contains direct support for the concept of mobile objects. Given this ability, you can have your Data Access layer (running on an application server) create a business object and load it with data from the database. You can then send that business object to the client machine (or web server), where the interface control code can use the object (as shown in Figure 1-16).

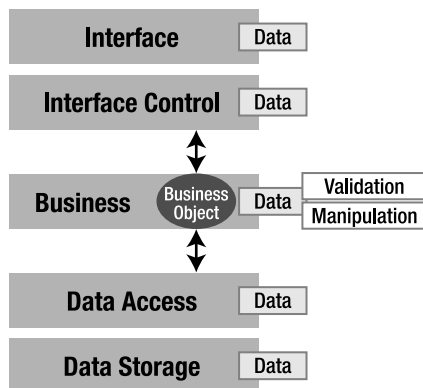


Figure 1-16. Using a business object to centralize business logic

In this architecture, smart data (in the form of a business object), rather than raw data, is sent to the client. Then the interface control code can use the same business logic as the data access code. This reduces maintenance, because you're not writing some business logic in the Data Access layer and some other business logic in the Interface Control layer. Instead, all the business logic is consolidated into a real, separate layer composed of business objects. These business objects will move across the network just like the DataSet did earlier, but they'll include the data *and* its related business logic—something the DataSet can't easily offer.

Note In addition, business objects will typically move across the network more efficiently than the DataSet. The approach in this book will use a binary transfer scheme that transfers data that is about 30 percent of the size of data transferred using the DataSet. Also, the business objects will contain far less metadata than the DataSet, further reducing the number of bytes transferred across the network.

Effectively, you're sharing the Business layer between the machine running the Data Access layer and the machine running the Interface Control layer. As long as there is support for mobile objects, this is an ideal solution: it provides code reuse, low maintenance costs, and high performance.

A New Logical Architecture

Being able to access the Business layer from both the Data Access layer and the Interface Control layer directly opens up a new way to view the logical architecture. Though the Business layer remains a separate concept, it's directly used by and tied into both the Interface Control and Data Access layers, as shown in Figure 1-17.

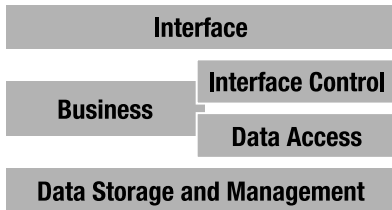


Figure 1-17. *The Business layer tied to the Interface Control and Data Access layers*

The Interface Control layer can interact directly with the objects in the Business layer, thereby relying on them to perform all validation, manipulation, and other processing of the data. Likewise, the Data Access layer can interact with the objects as the data is retrieved or stored.

If all the layers are running on a single machine (such as a smart client), then these parts will run in a single process and interact with each other with no network or cross-processing overhead. In more distributed physical configurations, the Business layer will run on both the client *and* the application server, as shown in Figure 1-18.

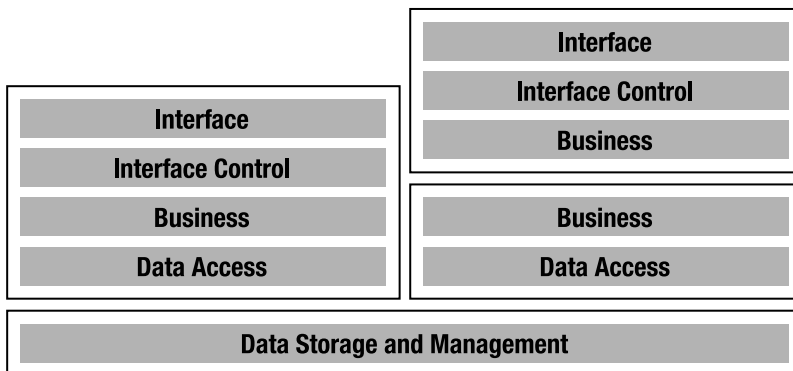


Figure 1-18. *Business logic shared between the Interface Control and Data Access layers*

Local, Anchored, and Mobile Objects

Normally, one might think of objects as being part of a single application, running on a single machine in a single process. A distributed application requires a broader perspective. Some of the objects might run only in a single process on a single machine. Others might run on one machine but might be called by code running on another machine. Still others might be mobile objects, moving from machine to machine.

Local Objects

By default, .NET objects are *local*. This means that ordinary .NET objects aren't accessible from outside the process in which they were created. Without taking extra steps in your code, it isn't possible to pass objects to another process or another machine (a procedure known as *marshaling*), either by value or by reference.

Anchored Objects

In many technologies, objects are always passed *by reference*. This means that when you “pass” an object from one machine or process to another, what actually happens is that the object remains in the original process, and the other process or machine merely gets a pointer, or reference, back to the object, as shown in Figure 1-19.

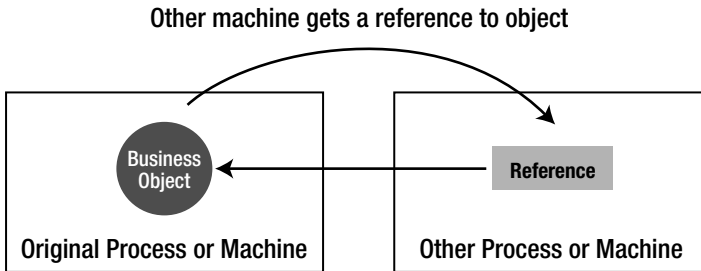


Figure 1-19. Calling an object by reference

By using this reference, the other machine can interact with the object. Because the object is still on the original machine, however, any property or method calls are sent across the network, and the results are returned back across the network. This scheme is useful only if the object is designed so it can be used with very few method calls; just one is ideal.

The recommended designs for Enterprise Services objects call for each method on the object to do all its work in a single method call for precisely this reason, thereby sacrificing “proper” object-oriented design in order to reduce latency. The same is effectively true for objects exposed to the network through WCF. Each method on a service object should do all its work, not relying on the client to have called other methods before or after calling this method.

These types of objects are stuck, or *anchored*, on the original machine or process where they were created. An anchored object never moves; it’s accessed via references.

In .NET, you can create an anchored object in a couple different ways. If you’re using WCF, the object will implement a service contract:

```
<ServiceContract(>> _
Public Interface IMyService
    <OperationContract(>> _
        Sub MyOperation()
    End Interface
Public Class MyServiceImplementation
    Implements IMyService
End Class
```

If you’re using the older .NET Remoting technology, you create an anchored object by having it inherit from `MarshalByRefObject`:

```
Public Class MyAnchoredClass
    Inherits MarshalByRefObject
End Class
```

Either way, the .NET Framework takes care of the details. Code running in another process or on another machine across the network can call the anchored object.

Mobile Objects

The concept of mobile objects relies on the idea that an object can be passed from one process to another, or from one machine to another, *by value*. This means that the object is physically copied from the original process or machine to the other process or machine, as shown in Figure 1-20.

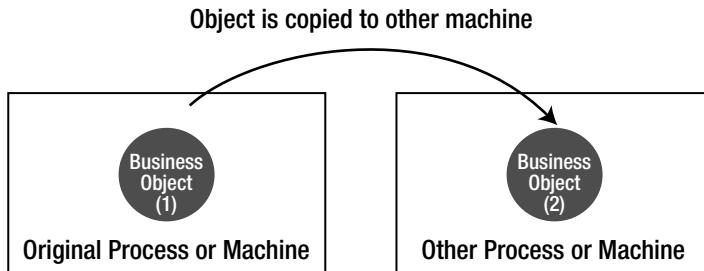


Figure 1-20. Passing a physical copy of an object across the network

The object starts on the left (instance 1) and is copied, or *cloned*, across the network to the right (instance 2).

Because the other machine gets a copy of the object, it can interact with the object locally. This means that there's effectively no performance overhead involved in calling properties or methods on the object—the only cost was in copying the object's data across the network in the first place.

Note One caveat here is that transferring a large object across the network can cause a performance problem. Returning a `DataSet` that contains a great deal of data can take a long time. This is true of all mobile objects, including business objects. You need to be careful in your application design in order to avoid retrieving very large sets of data.

Objects that can move from process to process or from machine to machine are *mobile objects*. Examples of mobile objects include the `DataSet` and the business objects created in this book. Mobile objects aren't stuck in a single place but can move to where they're most needed.

To create one in .NET, add the `<Serializable(>>` attribute to your class definition. You may also optionally implement the `ISerializable` interface. I'll discuss this further in Chapter 2, but the following illustrates the start of a class that defines a mobile object:

```

<Serializable(>> _
Public Class MyMobileClass
    Private _data As String
End Class
  
```

When using WCF, you might choose instead to use the `DataContract` and `DataMember` attributes:

```

<DataContract(>> _
Public Class MyMobileClass
    <DataMember(>> _
        Private _data As String
End Class
  
```


Either approach works, but the `<Serializable(>` attribute is often better for this scenario because it uses an opt-out approach. All fields of the object are serialized unless you explicitly prevent the serialization. The `DataContract` approach is opt-in, so you must remember to mark every field for serialization. Forget one, and you've just introduced a hard-to-find bug.

Again, the .NET Framework takes care of the details, so an object of this type can be simply passed as a parameter to a method call or as the return value from a function. The object will be copied from the original machine to the machine where the method is running.

It is important to understand that the *code* for the object isn't moved across the network automatically. Before an object can move from machine to machine, both machines must have the .NET assembly containing the object's code installed. Only the object's serialized data is moved across the network by .NET. Installing the required assemblies is often handled by ClickOnce or other .NET deployment technologies.

When to Use Which Mechanism

The .NET Framework supports all the mechanisms just discussed, so you can choose to create your objects as local, anchored, or mobile, depending on the requirements of your design. As you might guess, there are good reasons for each approach.

WPF, Windows Forms, and ASP.NET objects are all local—they're inaccessible from outside the processes in which they were created. The assumption is that other applications shouldn't be allowed to just reach into your program and manipulate your interface objects.

Anchored objects are important because they will always run on a specific machine. If you write an object that interacts with a database, you'll want to ensure that the object will always run on a machine that has access to the database. Because of this, anchored objects are typically used on application servers.

Many business objects, on the other hand, will be more useful if they *can* move from the application server to a client or web server, as needed. By creating business objects as mobile objects, you can pass smart data from machine to machine, thereby reusing your business logic anywhere the business data is sent.

Typically, anchored and mobile objects are used in concert. Later in the book, I'll show how to use an anchored object on the application server to ensure that specific methods are run *on that server*. Then mobile objects will be passed as parameters to those methods, which will cause those mobile objects to move from the client to the server. Some of the anchored server-side methods will return mobile objects as results, in which case the mobile object will move from the server back to the client.

Passing Mobile Objects by Reference

There's a piece of terminology here that can get confusing. So far, I've loosely associated anchored objects with the concept of *passing by reference*, and mobile objects as being *passed by value*. Intuitively, this makes sense, because anchored objects provide a reference, though mobile objects provide the actual object (and its values). However, the terms *by reference* and *by value* have come to mean other things over the years.

The original idea of passing a value by reference was that there would be just one set of data—one object—and any code could get a reference to that single entity. Any changes made to that entity by any code would therefore be immediately visible to any other code.

The original idea of passing a value by value was that a copy of the original value would be made. Any code could get a copy of the original value, but any changes made to that copy weren't reflected in the original value. That makes sense, because the changes were made to a copy, not to the original value.

In distributed applications, things get a little more complicated, but the previous definitions remain true: an object can be passed by reference so that all machines have a reference to the same object on a server. And an object can be passed by value so that a copy of the object is made. So far, so good. However, what happens if you mark an object as `<Serializable()>` or `<DataContract()>` (that is, mark it as a mobile object) and then *intentionally* pass it by reference? It turns out that the object is passed by value, but the .NET Framework attempts to provide the illusion that the object was passed by reference.

To be more specific, in this scenario, the object is copied across the network just as if it were being passed by value. The difference is that the object is then returned to the calling code when the method is complete, and the reference to the original object is replaced with a reference to this new version, as shown in Figure 1-21.

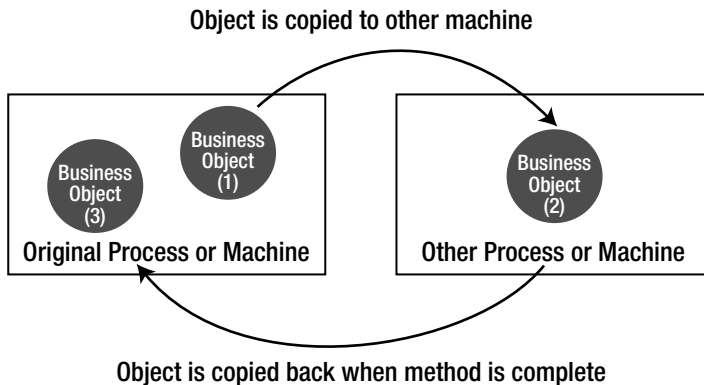


Figure 1-21. Passing a copy of the object to the server and getting a copy back

The original object (instance 1) starts on the left and is copied (cloned) to the right (instance 2). The right-hand process or machine has full access to a local copy of the object and can interact with it freely. When that's done, the object is "returned" to the left, meaning it is copied back across the network into a new object on the left (instance 3).

This is potentially very dangerous, since *other* references to the original object continue to point to that original object—only this one particular reference is updated. You can end up with two different versions of the same object on the machine, with some references pointing to the new one and some to the old one.

Note If you pass a mobile object by reference, you must always make sure to update *all* references to use the new version of the object when the method call is complete.

You can choose to pass a mobile object by value, in which case it's passed one way: from the caller to the method. Or you can choose to pass a mobile object by reference, in which case it's passed two ways: from the caller to the method and from the method back to the caller. If you want to get back any changes the method makes to the object, use *by reference*. If you don't care about or don't want any changes made to the object by the method, use *by value*.

Note that passing a mobile object by reference has performance implications—it requires that the object be passed back across the network to the calling machine, so it's slower than passing by value.

Complete Encapsulation

At this point, your imagination should be engaged by the potential of mobile objects. The flexibility of being able to choose between local, anchored, and mobile objects is very powerful, and it opens up new architectural approaches.

I've already discussed the idea of sharing the Business layer across machines, and it's probably obvious that the concept of mobile objects is exactly what's needed to implement such a shared layer. But what does this all mean for the *design* of the layers? In particular, given a set of mobile objects in the Business layer, what's the impact on the Interface Control and Data Access layers with which the objects interact?

Impact on the Interface Control Layer

What it means for the Interface Control layer is simply that the business objects will contain all the business logic. The interface developer can code each form, page, service, or workflow activity using the business objects, thereby relying on them to perform any validation or manipulation of the data. This means that the interface code can focus entirely on displaying the data, interacting with the user, and providing a rich, interactive experience.

More important, because the business objects are mobile, they'll end up running in the same process as the interface control code. Any property or method calls from the interface control code to the business object will occur locally without network latency, marshaling, or any other performance overhead.

Impact on the Data Access Layer

A traditional Data Access layer consists of a set of methods or services that interact with the database and with the objects that encapsulate data. The data access code itself is typically outside the objects, rather than being encapsulated within the objects. This, however, breaks encapsulation, since it means that the objects' data must be externalized to be handled by the data access code.

The framework created in this book allows for the data access code to be encapsulated within the business objects or externalized into a separate set of objects. As you'll see in Chapter 7, there are both performance and maintainability benefits to including the data access code directly inside each business object. However, there are security and manageability benefits to having the code external.

Either way, the concept of a Data Access layer is of key importance. Maintaining a strong logical separation between the data access code and business logic is highly beneficial, as discussed earlier in this chapter. Obviously, having a totally separate set of data access objects is one way to clearly implement a Data Access layer.

Architectures and Frameworks

The discussion so far has focused mainly on architectures: logical architectures that define the separation of responsibilities in an application, and physical architectures that define the locations where the logical layers will run in various configurations. I've also discussed the use of object-oriented design and the concepts behind mobile objects.

Although all of these are important and must be thought through in detail, you really don't want to have to go through this process every time you need to build an application. It would be preferable to have the architecture and design solidified into reusable code that could be used to build all your

applications. What you want is an *application framework*. A framework codifies an architecture and design in order to promote reuse and increase productivity.

The typical development process starts with requirements gathering and analysis, followed by a period of architectural discussion and decision making. Next comes the application design: first the low-level concepts to support the architecture and then the business-level concepts that actually matter to the end users. With the design completed, developers typically spend a fair amount of time implementing the low-level functions that support the business coding that comes later.

All of the architectural discussions, decision making, designing, and coding can be a lot of fun. Unfortunately, it doesn't directly contribute anything to the end goal of writing business logic and providing business functionality. This low-level supporting technology is merely "plumbing" that must exist in order to create actual business applications. It's an overhead that in the long term you should be able to do once and then reuse across many business application-development efforts.

In the software world, the easiest way to reduce overhead is to increase reuse, and the best way to get reuse out of an architecture (both design and coding) is to codify it into a framework.

This doesn't mean that *application* analysis and design are unimportant—quite the opposite! People typically spend far too little time analyzing business requirements and developing good application designs to meet those business needs. Part of the reason is that they often end up spending substantial amounts of time analyzing and designing the plumbing that supports the business application, and then run out of time to analyze the business issues themselves.

What I'm proposing here is to reduce the time spent analyzing and designing the low-level plumbing by creating a framework that you can use across many business applications. Is the framework created in this book ideal for every application and every organization? Certainly not! You'll have to take the architecture and the framework and adapt them to meet your organization's needs. You may have different priorities in terms of performance, scalability, security, fault tolerance, reuse, or other key architectural criteria. At the very least, though, the remainder of this book should give you a good start on the design and construction of a distributed, object-oriented architecture and framework.

Conclusion

In this chapter, I've focused on the theory behind distributed systems—specifically, those based on mobile objects. The key to success in designing a distributed system is to keep clear the distinction between a logical architecture and a physical architecture.

Logical architectures exist to define the separation between the different types of code in an application. The goal of a good logical architecture is to make code more maintainable, understandable, and reusable. A logical architecture must also define enough layers to enable any physical architectures that may be required.

Physical architectures define the machines on which the application will run. An application with several logical layers can still run on a single machine. You also might configure that same logical architecture to run on various client and server machines. The goal of a good physical architecture is to achieve the best trade-off between performance, scalability, security, and fault tolerance within your specific environment.

The trade-offs in a physical architecture for a smart client application are very different from those for a web application. A Windows application will typically trade performance against scalability, and a web application will typically trade performance against security.

In this book, I'll be using a 5-layer logical architecture consisting of the Interface, Interface Control, Business, Data Access, and Data Storage and Management layers. Later in the book, I'll show you how to use this architecture to create Windows, web, and service-oriented applications, each with a different physical architecture. The next chapter will start the process of designing the framework that will make this possible.

