expert one-on-one
Visual Basic .NET Business Objects

Rockford Lhotka

Apress™

expert one-on-one

Visual Basic .NET Business Objects

expert one-on-one Visual Basic .NET Business Objects

ISBN (pbk): 1-59059-145-3

Printed and bound in the United States of America 2345678910

Technical Reviewers: Mark Horner, David Schultz Donald Xie

Editorial Directors: Dan Appleman, Gary Cornell, Martin Streicher, Jim Sumser, Karen Watterson, John Zukowski

Project Manager: Nicola Phillips

Proofreader: Helena Sharman

Production Editors: Sarah Hall, Paul Grove

Indexer: Martin Brooks

Artist and Cover Designer: Kurt Krames

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, email `orders@springer.de`, or visit `http://www.springer.de`.

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

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

Distributed architecture

Object-oriented design and programming are big topics – there are entire books devoted solely to the process of object-oriented design, and other books devoted to using object-oriented programming in various languages and on various programming platforms. My focus in this book is not to teach the basics of object-oriented design or programming, but rather to show how to 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 around distributed computing as it relates to object-oriented development. We'll cover a number of topics, including:

- How logical n-tier architectures help address reuse and maintainability
- How physical n-tier architectures impact performance, scalability, security and fault tolerance
- Data-centric vs. object-oriented application models
- How object-oriented models help increase code reuse and application maintainability
- Effective use of objects in a distributed environment, including the concepts of anchored and unanchored 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, we'll be exploring an n-tier architecture that may be physically distributed across multiple machines. We'll also be using object-oriented design and programming techniques to implement a framework supporting this architecture. With that done, we'll create sample applications to demonstrate how the architecture and the framework support our 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. While the entire application *might* run on a single machine, it's more likely that the application will run on a web server, or be split between an intelligent client and an application server. Given these varied physical environments, we are faced with questions such as:

- Where do the objects reside?
- Are the objects designed to maintain state, or should they be stateless?
- How do we handle object-to-relational mapping when we retrieve or store data in the database?
- How do we manage database transactions?

Before we get into discussing some answers to these questions, it's important that we fully understand the difference between a **physical architecture** and a **logical architecture**. After that, we'll define objects and distributed objects, and see how they fit into the architectural discussion.

When most people talk about n-tier applications, they're talking about physical models where 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 is not 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 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-tier architecture. The latter 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 a UI tier, a business tier, and a data tier that may exist on a single machine, or on three separate machines – the logical architecture doesn't define those details.

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

When you start looking around, the sad reality is that many applications have no clearly defined logical architecture – the logical architecture merely defaults to the number of physical tiers. This lack of formal logical design causes problems, since it reduces flexibility. If we design a system 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 we start by creating a *logical* architecture of three tiers, we can switch more easily between one, two, or three physical tiers later on.

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-tier architecture. A properly designed logical n-tier 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's a risk of damaging the things that would have been improved had the job been done well.

Complexity

As experienced designers and developers, we 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 (logical and/or physical) 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 only simplify the process for large applications or complex environments. They can easily complicate matters if all we'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 is 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 is not complex

Something to remember is that even a small application is likely to grow, and even a simple environment will often become more complex over time. The more successful our 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, expecting and planning for growth.

This discussion illustrates why n-tier applications are viewed as relatively complex. There are a lot of factors, technical and non-technical, that must be taken into account. Unfortunately, it is not possible to say definitively when n-tier does and doesn't fit. In the end, it is a judgment call that we, as architects of our applications, must make, based on the factors that affect our particular organization, environment, and development team.

Relationship between logical and physical models

Architectures such as .NET's forebear, Windows DNA, represent a merger of logical and physical models. Such mergers seem attractive because they appear so simple and straightforward, but typically they're not good in practice – they can lead people to design applications using a logical or physical architecture that's not best suited to their needs.

> *To be fair, Windows DNA didn't* mandate *that the logical and physical models be the same. Unfortunately, almost all of the printed material (even the mouse mats) surrounding Windows DNA included diagrams and pictures that illustrated the 'proper' Windows DNA implementation as an intertwined blur of physical and logical architecture. While some experienced architects were able to separate the concepts, many more didn't, and created some horrendous results.*

The logical model

When we'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 tiers so that we have flexibility in choosing a physical architecture later on.

Traditionally, we'd devise at least a 3-tier logical model that separates the interface, the logic, and the data management portions of the application. Today that is rarely sufficient, because the 'interface' tier is often physically split into two parts (browser and web server), and the 'logic' tier is often physically split between a client or web server and an application server. Additionally, there are various application models that break the traditional business tier up into multiple parts – model-view-controller and façade-data-logic being two of the most popular at the moment.

This means that our logical tiers are governed by the following rules:

- The logical architecture includes tiers to organize our components into discrete roles
- The logical architecture must have at least as many tiers as our anticipated physical deployment

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

The physical model

By ensuring that the logical model has enough tiers to give us flexibility, we can configure our application into an appropriate physical architecture that will depend on our performance, scalability, fault-tolerance, and security requirements. The more physical tiers we include, the worse our performance will be – but we have 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 we 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 we add load (such as increased users) 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 we decide that we need to support multiple users, we 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 we've introduced network latency and network contention too. To overcome this problem, we could put the data into a managed environment such as SQL Server or Oracle. This will help to reduce data contention, but we'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 are being used by other clients. We can reduce this load by shifting some of the work off to another server. An **application server** such as MTS or COM+ (sometimes referred to as Enterprise Services in .NET) 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.

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 given user, the system gets slower – but we are 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 our database or application server is underpowered, they can become so slow that data transfer across the network is not an issue. Likewise, if our application does extremely intense calculations and our 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 if we narrow our discussion solely to consider how it's affected by physical n-tier decisions, it becomes more approachable. We find that we're not talking about authentication or authorization as much as we're talking about controlling physical access to the machines on which portions of our application will run. The number of physical tiers in an application has no impact on whether we can authenticate or authorize users, but we *can* use physical tiers to increase or decrease physical access to the machines where our application executes.

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, while a Web Forms 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 our application? The surface area can be defined in terms of **domains of trust**.

Security and internal applications

Internal applications are totally encapsulated within our domain of trust – the client and all servers are running in a trusted environment. This means that virtually every part of our application is exposed to a potential hacker (assuming that the hacker can gain physical access to a machine on our network in the first place). In a typical organization, a hacker 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 LAN.

> *Obviously, we do* have *security – we typically use Windows domain or Active Directory security on our clients and servers, for instance – but there's nothing stopping someone from attempting to communicate directly with any of these machines. What we're talking about here is* access*, and within a typical network, we have access to all machines.*

Because the internal environment is so exposed to start with, security should have little impact on our decisions regarding the number of physical tiers for our application. Increasing or decreasing the number of tiers will rarely have much impact on a hacker's ability to compromise the application from a client workstation on our LAN.

An exception to this rule comes when someone can use our own web services or remoting services to access our servers in invalid ways. This problem was particularly acute with DCOM, because there were browsers that could be used by end users to locate and invoke server-side services. Thanks to COM, users could use Excel to locate and interact with server-side COM components, bypassing the portions of our application that were *supposed* to run on the client. This meant that we were vulnerable to power users who could use our components in ways we never imagined!

> *The problem is likely to transfer to web services in the near future, as new versions of Microsoft Office and other end-user tools gain web service browsers. We can then expect to find power users writing macros in Microsoft Excel to invoke our web services in ways we never expected.*

The technology we'll be using in this book is .NET remoting, which is not geared toward the same ease of use as web services, and it's unlikely that end users will have browsers to locate remoting services. Even so, we'll be designing our remoting services to prevent casual usage of our objects, even if a power user were to gain access to the service from some future version of Microsoft Excel!

In summary, while security should not cause us to increase or decrease the number of physical tiers for internal applications, it *should* inform our design choices when we expose services from our server machines.

Security and external applications

For external applications, things are entirely different. To start with, we assume that there are at least two tiers: the client workstation is separate from any machines physically running within our environment. Typically, the client workstations are on the other side of a firewall from any of our servers, and we control the specific IP ports by which they gain entry to our network.

This means that the client workstations are outside our domain of trust, which in turn means that we must assume they are compromised and potentially malicious. If we actually run any code on those clients, we must assume that it ran incorrectly or not at all – in other words, we must completely validate any input from the client as it enters our domain of trust, even if we put code into the client to do the validation.

In many web applications, for instance, we'll include script code that executes on the browser to validate user input. When the user posts that data back to our Web Form, we must revalidate the data, because we must assume that the user somehow defeated or altered the client-side validation, and is now providing us with invalid data.

> *I've had people tell me that this is an overly paranoid attitude, but I've been burned this way too many times. Any time we are exposing an interface (Windows, web, XML, etc.) such that it can be used by clients outside our control, we* must *assume that the interface will be misused. Often, this misuse is unintentional – someone wrote a macro to automate data entry rather than doing it by hand – but the end result is that our application fails unless we completely validate the input as it enters our domain of trust.*

The ideal in this case is to expose only one server (or one type of server – a web server, say) to clients that are outside our domain of trust. That way, we only have one 'port of entry', at which we can completely control and validate any inbound data or requests. It also reduces the hacker footprint by providing only one machine with which a hacker can interact. At this stage, we've only dictated two physical tiers: the client, and our server.

Many organizations take this a step further, and mandate there to be a second firewall behind which all data must reside. Only the web server can sit between the two firewalls. The idea is that the second firewall prevents a hacker from gaining access to any sensitive data, even if they breach the first firewall and take control of the web server. Typically, a further constraint in configurations like this is that the web server can't interact directly with database servers. Instead, the web server must communicate with an application server (which is behind the second firewall), and that server communicates with the database server.

There's some debate as to how much security is gained through this approach, but it's a common arrangement. What it means to *us* is that we now have a minimum of four tiers: the client, the web server, an application server, and a data server – and as we discussed earlier, the more physical tiers we have, the worse our performance will be. As a general rule, switching from a 3-tier web model (client, web server, database server) to this type of 4-tier web model (client, web server, application server, database server) will result in a 50% performance reduction.

Some of this performance hit can be mitigated by special network configurations – using dual NICs and special routing for the servers, for example – but the fact remains that there's a substantial impact. That second firewall had better provide a lot of extra security, because we're making a big sacrifice in order to implement it.

Fault tolerance

Fault tolerance is achieved by identifying points of failure and providing redundancy. Typically, our applications have numerous points of failure. Some of the most obvious are:

- The network feed to our building or data center
- The power feed to our building or data center
- The network feed and power feed to our ISP's data center
- The primary DNS host servicing our domain
- Our firewall
- Our web server
- Our application server
- Our database server
- Our internal LAN

To achieve high levels of fault tolerance we need to ensure that if any one of these fails, some system will instantly kick in and fill the void. If our power goes out, a generator kicks in. If our network feed is cut by a bulldozer, we have a second network feed coming in from the other side of our 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 due to 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 due to 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 our application architecture. Adding redundant servers, on the other hand, *will* affect our n-tier application architecture – or at least, our application design. Each time we add a physical tier to our n-tier model, we need to ensure that we can add redundancy to the servers in that tier. The more physical tiers, the more redundant servers we need to configure and maintain. Thus, adding a tier always means adding at least *two* servers to our infrastructure.

Not only that, but to achieve fault tolerance through redundancy, all servers in a tier must also be identical at all times. In other words, at no time can a user be tied to a specific server – in other words, no 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, and we have lost fault tolerance (for that user, at least).

Achieving a high degree of fault tolerance is not 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 our architecture can assist in this process by reducing the number of tiers that must be made redundant.

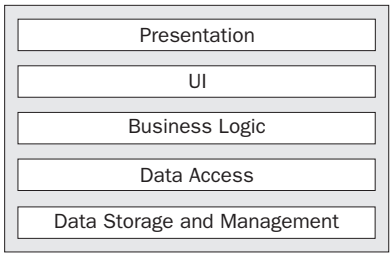
Ultimately, the number of physical tiers in our architecture is a trade-off between performance, scalability, security, and fault tolerance. Furthermore, the optimal configuration for a web application is not the same as that for an intranet application with intelligent client machines. If the framework we're going to create is to have any hope of broad appeal, we need flexibility in the physical architecture so that we can support web and intelligent clients effectively, and provide both with optimal performance and scalability.

A 5-tier logical architecture

In this book, I'll be exploring a *5*-tier logical architecture, and showing how we can implement it using object-oriented concepts. Once we've created it, we'll configure the logical architecture into various physical architectures to achieve optimal results for Windows Forms, Web Forms, and web services interfaces.

> *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, as throughout the book, is to present a coherent, distributed, object-oriented architecture that supports Windows, web, and web services interfaces.*

In our framework, the logical architecture comprises the five tiers shown in the following figure:



Remember that the benefit of a logical n-tier 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 tiers more carefully.

Presentation

At first, it may not be clear why I've separated presentation from the user interface (UI). Certainly, from a Windows perspective, presentation and UI are one and the same: some GUI forms with which the user can interact. From a web perspective (or from that of terminal-based programming), however, the distinction is probably quite clear. The browser (or a terminal) merely presents information to the user, and collects user input. All of the actual interaction logic – the code we write to *generate* the output, or to *interpret* user input – runs on the web server (or mainframe), and not on the client machine.

Knowing that our logical model must support both intelligent 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 presentation *will* be physically separate from the user interface logic. To accommodate this separation, we will need to design our applications around this concept.

> *The types of presentation tiers continue to multiply, and each comes with a new and relatively incompatible technology with which we must work. It's virtually impossible to create a programming framework that entirely abstracts presentation concepts. Because of this, our architecture and framework will merely* support the creation *of varied presentations, not automate or simplify them. Instead, our focus will be on simplifying the other tiers in the architecture, where technology is more stable.*

UI

Now that we understand the distinction between presentation and UI, the latter's purpose is probably fairly clear. This tier includes the logic to decide what the user sees, the navigation paths, and how to interpret user input. In a 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 tier.

In many applications, the UI code is very complex. For a start, it must respond to the user's requests in a non-linear fashion. (We have little control over how users might click on controls, or enter or leave our forms or pages.) The UI code must also interact with logic in the business tier to validate user input, to perform any processing that's required, or to do any other business-related action.

Basically, what we're talking about here is writing UI code that accepts user input and then provides it to the business logic, where it can be validated, processed, or otherwise manipulated. The UI code must then respond to the user by displaying the results of its interaction with the business logic. Was the user's data valid? If not, what was wrong with it? And so forth.

In .NET, our UI code is almost always event-driven. Windows Forms code is all about responding to events as the user types and clicks on our form, and Web Forms code is all about responding to events as the browser round-trips the user's actions back to the web server. While both Windows Forms and Web Forms technologies make heavy use of objects, the code that we typically write into our UI is not object-oriented as much as it is procedural and event-based.

That said, there is great value in creating frameworks and reusable components to support a particular type of UI. If we're creating a Windows Forms UI, we can make use of visual inheritance and other object-oriented techniques to simplify the creation of our forms. If we're creating a Web Forms UI, we can use ASCX user controls and custom server controls to provide reusable components that simplify page development.

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

Business logic

Business logic includes all business rules, data validation, manipulation, processing, and security for our application. One definition from Microsoft is, *"The combination of validation edits, logon verifications, database lookups, policies, and algorithmic transformations that constitute an enterprise's way of doing business."*

The business logic *must* reside in a separate tier from the UI code. I believe that this particular separation is the most important if we want to gain the benefits of increased maintainability and reusability for our code. This is because any business logic that creeps into the UI tier will reside within a *specific* UI, and will not be available to any other UIs that we might later create.

Any business logic that we write into (say) our Windows UI is useless to a web or web service UI, and must therefore be written into those as well. This instantly leads to duplicated code, which is a maintenance nightmare. Separation of these two tiers can be done through techniques such as clearly defined procedural models, or object-oriented design and programming. In this book, we'll be applying object-oriented concepts: encapsulating business data and logic in a set of objects is a powerful way to accomplish separation of the business logic from the user interface.

Data access

Data access code interacts with the data management tier to retrieve, update, and remove information. The data access tier 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 tier for much the same reason that we split presentation from the user interface. In some cases, data access will occur on a machine that's physically separate from the one where the UI 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 UI) in order to improve performance or fault tolerance.

> *It may sound odd to say that putting the data access tier on the same machine as our business logic can* increase *fault tolerance, but consider the case of web farms, where each web server is identical to all the others. By putting the data access code on the web servers, we provide automatic redundancy of the data access tier along with the business logic and UI tiers.*
>
> *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, so it's not something that should be done lightly.*

By logically defining data access as a separate tier, we enforce a separation between the business logic and how we interact with a database (or any other data source). This separation gives us 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 we may need to switch 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 we need to rewrite our data access code to keep up (remember DAO, RDO, ADO 1.0, ADO 2.0, and now ADO.NET?). By isolating the data access code into a specific tier, we limit the impact of these changes to a smaller part of our application.

Data access mechanisms are typically implemented as a set of services, with each service being a procedure that's called by the business logic to create, retrieve, update, or delete data. While these services are often constructed using objects, it's important to recognize that the designs for an effective data access tier 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.

> *If we're using an object database instead of a relational database, then of course our data access code may be very object-oriented. Few of us get such an opportunity, however, since almost all data is stored in relational databases.*

Sometimes, the data access tier 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 tier is more complex, providing a more abstract or even metadata-driven way to get at data. In these cases, the data access tier can contain a lot of complex code to provide this more abstract data access scheme. The framework we'll create in this book will work directly against ADO.NET, but you could also use a metadata-driven data access layer if you prefer.

Another common role for the data access tier 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 our relational model and getting it into the object-oriented model is called **object-relational mapping**, and we'll have more to say on the subject in Chapter 2.

Data storage and management

Finally, we have the data storage and management tier. Database servers such as SQL Server or Oracle often handle these tasks, but increasingly other applications may provide this functionality too, via technologies such as web services.

What's key about this tier is that it handles the physical creation, retrieval, update, and deletion of data. This is different from the data access tier, which *requests* the creation, retrieval, update, and deletion of data. In the data management tier, we actually *implement* these operations within the context of a database or a set of files, etc.

The data management tier is invoked by our business logic (via the data access tier), but the former 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 management tier will include business logic that we also implement in our business logic tier. This time, the replication is unavoidable, because relational databases are designed to enforce relational integrity – and that's just another form of business logic.

In any case, whether we're using stored procedures in SQL Server, or SOAP calls to another application, data storage and management is typically handled by creating a set of services or procedures that can be called as needed. Like the data access tier, it's important to recognize that the designs for data storage and management are typically very procedural.

The following table summarizes the five tiers and their roles:

Tier	Roles
Presentation	Renders display and collects user input.
UI	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 ADO.NET), databases, and data structures.
Data Storage and Management	Physically creates, retrieves, updates, and deletes data in a persistent data store.

Everything we've talked about to this point is part of a *logical* architecture. Now let's move on and see how it can be applied in various *physical* configurations.

Applying the logical architecture

Given this 5-tier logical architecture, we should be able 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.

In this discussion, we're assuming that we have total flexibility to configure what logical tier runs where. In some cases, there are technical issues that prevent physical separation of some tiers. Fortunately, there are fewer such issues with the .NET Framework than there were with COM-based technologies.

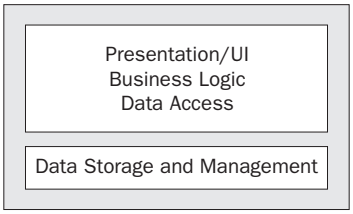
There are a few physical configurations that I want to discuss in order to illustrate how our logical model works. These are common and important setups that most of us encounter on a day-to-day basis.

Optimal performance intelligent 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 application often run in standalone environments. However, we still want the benefits of the logical n-tier architecture in terms of maintainability and code reuse.

It probably goes without saying that if we want to, we can install everything on a single client workstation. An optimal performance intelligent client is usually implemented using Windows Forms for the presentation and UI, with the business logic and data access code running in the same process and talking to a JET or MSDE (Microsoft SQL Server Desktop Engine) database. The fact that the system is deployed on a single physical tier doesn't compromise the logical architecture and separation.

I think it's very important to remember that n-tier systems can run on a single machine in order to support the wide range of applications that require standalone machines. It's also worth pointing out that this is basically the same as 2-tier, 'fat client' physical architecture – the only difference in that case is that the data storage and management tier would be running on a central database server, such as SQL Server or Oracle:

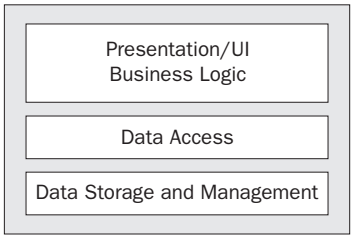


Other than the location of the data storage, this is identical to the single-tier configuration, and typically the switch from single-tier to 2-tier revolves around little more than changing the database configuration string for ADO.NET.

High scalability intelligent client

Single-tier configurations are good for standalone environments, but they don't scale well. To support multiple users, we often 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, we can trade performance to gain scalability by moving the data access tier to a separate machine. Single- or 2-tier configurations give the best performance, but they don't scale as well as a 3-tier configuration would do. A good rule of thumb is that if you have more than 50-100 concurrent users, you can gain by making use of a separate server to handle the data access tier:



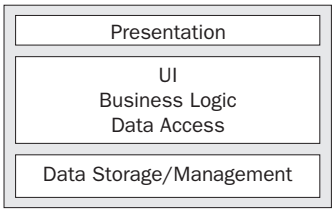
By doing this, we can centralize all access to the database on a single machine. In .NET, if the connections to the database for all our users are made using the same user ID and password, we'll get the benefits of **connection pooling** for all our users. What this means immediately is that there will be far fewer connections to the database than there would be if each client machine connected directly. The actual reduction depends on the specific application, but we're often looking at supporting 150-200 concurrent users with just 2 or 3 database connections!

Of course, all user requests now go across an extra network hop, causing increased latency (and therefore decreased performance). This performance cost translates into a huge scalability gain, however, since this architecture can handle many more concurrent users than a 2-tier physical configuration. If well designed, such an architecture can support *thousands* of concurrent users with adequate performance.

Optimal performance web client

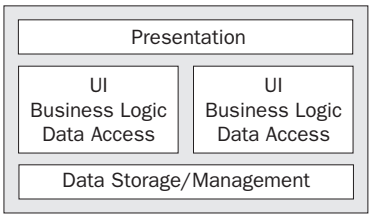
As with a Windows Forms application, we get the best performance from a web-based application by minimizing the number of physical tiers. However, the tradeoff in a web scenario is different: in this case, we can improve performance and scalability at the same time, but at the cost of security, as we'll see shortly.

To get optimal performance in a web application, we want to run most of our code in a single process on a single machine, as shown in the following figure:



The presentation tier *must* be physically separate, because it's running in a browser, but the UI, the business logic, and the data access tier can all run on the same machine, in the same process. In some cases, we might even put the data management tier on the same physical machine, though this is only suitable for smaller applications.

This minimizes network and communication overhead, and optimizes performance. We also get very good scalability, since the web server can be part of a web farm in which all the web servers are running the same code:



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

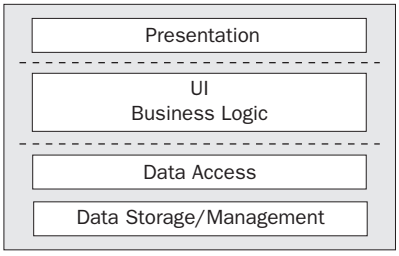
> *With COM-based technologies such as ASP and Visual Basic 6, this configuration was problematic, because running COM components in the same process as ASP pages had drawbacks in terms of the manageability and stability of the system. Running the COM components in a COM+ Server Application addressed the stability issues, but at the cost of performance. These issues have been addressed in .NET, however, so this configuration is highly practical when using ASP.NET and other .NET components.*

Unless we notice that our database server is getting overwhelmed with connections from the web servers in our web farm, there will rarely be gains to be made in scalability by using a separate application server. If we *do* decide that we need a separate application server, we must realize that we'll reduce performance because we're adding another physical tier. (Hopefully, we'll gain scalability, since the application server can consolidate database connections across all the web servers.) We must also consider fault tolerance in this case, because we may need redundant application servers to avoid a point of failure.

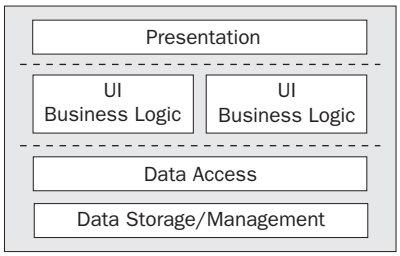
Another reason for implementing an application server is to increase security, and that's the topic of the next section.

High security web client

As we discussed in the earlier section on security, there will be many projects in which it is dictated that a web server can never talk directly to a database. The web server must run in a 'demilitarized zone' or 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. This is illustrated by the following figure, where the dashed lines represent the firewalls:



By splitting out the data access tier and running it on a separate application server, we are able to increase the security of the application. However, this comes at the cost of performance – as we discussed earlier, this configuration will typically cause a performance degradation of around 50%. Scalability, on the other hand, is fine: as with the first web configuration, we can achieve it by implementing a web farm in which each web server runs the same UI and business logic code.



The way ahead

After we've implemented the framework to support this 5-tier architecture, we'll create a sample application with three different interfaces: Windows, web, and web services. This will give us the opportunity to see first hand how our framework supports the following models:

- High scalability intelligent client
- Optimal performance web client
- Optimal performance web client (variation to support web services)

Due to the way we'll implement our framework, switching to any of the other models we've just discussed will just require some configuration file changes, meaning that we can easily adapt our application to any of the physical configurations without having to change our code.

Managing business logic

At this point, you should have a good understanding of logical and physical architectures, and we've seen how a 5-tier logical architecture can be configured into various n-tier physical architectures. Now, in one way or another, all of these tiers will use or interact with our application's data. That's obviously the case for the data management and data access tiers, but the business logic tier must validate, calculate, and manipulate data; the UI transfers data between the business and presentation tiers (often performing formatting or using the data to make navigational choices); and the presentation tier displays data to the user, and collects new data as it is entered.

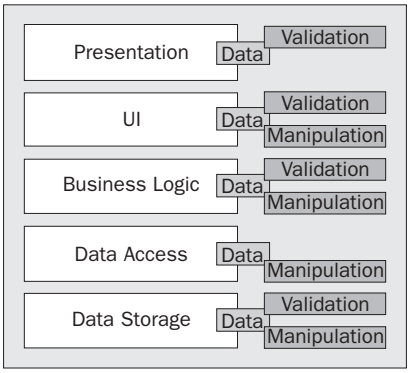
Similarly, it would be nice if all of our business logic would exist in the business logic tier, but in reality this is virtually impossible to achieve. In a web-based UI, we often include validation logic in the presentation tier, so that the user gets a more interactive experience in the browser. Similarly, most databases enforce referential integrity, and often some other rules too. Furthermore, the data access tier will very often include 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 tiers.

There is one key truth here that is important: for each piece of application data, there is 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 – a validation rule, for example, can be applied in the presentation tier and then reapplied in the UI tier or business logic tier before being sent to the database for storage. In some cases, the database will include code to recheck the value as well.

Let's look at some of the more common options. We'll start with three popular (but flawed) approaches, and then discuss a compromise solution that is enabled through the use of distributed objects such as the ones we'll be supporting in the framework we'll create later in the book.

Potential business logic locations

The following figure 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.



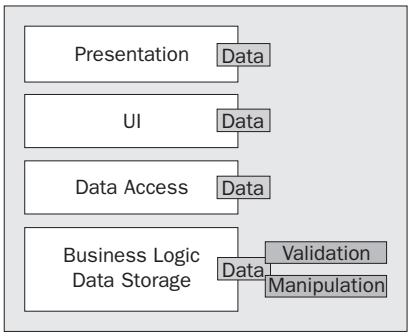
We put business logic in a web presentation tier to give the user a more interactive experience – and we put it into a Windows UI for the same reason. We recheck the business logic in the web UI (on the web server) because we don't trust the browser. And the database administrator puts the logic into the database (via stored procedures) because they don't trust any application developer!

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, and so we're constantly debugging and maintaining our code in a near-futile effort to make all of these redundant bits of logic agree with each other.

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

Business logic in the data management tier

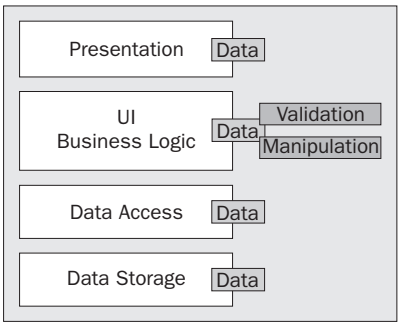
The classic approach is to put all logic into the database as the single, central repository. The presentation and UI then allow the user to enter absolutely anything (since any validation would be redundant), and the business logic tier is essentially gone – it's merged into the database. The data access tier does nothing but move the data into and out of the database.



The advantage of this approach is that the logic is centralized, but the drawbacks are plentiful. For a start, 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, since it's the only thing doing any actual work – and we have to write all of our business logic in SQL!

Business logic in the UI tier

Another common approach is to put all of the business logic into the UI. The data is validated and manipulated in the UI, and the data storage tier just stores the data. This approach is very common in both Windows and web environments, and has the advantage that the business logic is centralized into a single tier (and of course we can write our business logic in a language such as VB.NET or C#).

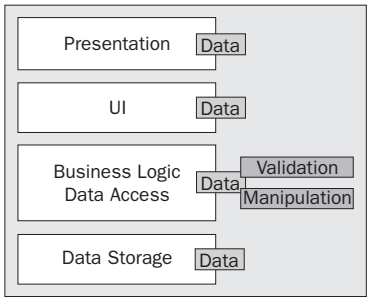


Unfortunately, in practice, the business logic ends up being scattered throughout the UI and intermixed with the UI code itself, decreasing readability and making maintenance more difficult. Even more importantly, business logic in one form or page is not reusable when we create subsequent pages or forms that use the same data. Furthermore, in a web environment, this architecture also leads to a totally non-interactive user experience, since no validation can occur in the browser. The user must transmit their data to the web server for any validation or manipulation to take place.

> *ASP.NET Web Forms' validation controls at least allow us to perform basic data validation in the UI, with that validation automatically extended to the browser by the Web Forms technology itself. While not a total solution, this is a powerful feature that does help.*

Business logic in the middle (business/data access) tier

Still another option is the classic UNIX client-server approach, where the business logic and data access tiers are merged, keeping the presentation, UI, and data storage tiers as 'dumb' as possible.

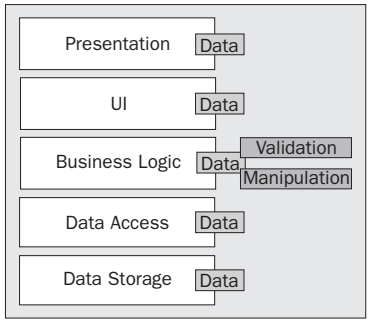


Unfortunately, once again, this approach falls foul of the non-interactive user experience problem: the data must be round-tripped to the data access tier for any validation or manipulation. This is especially problematic if the data access tier is running on a separate application server, since we're then faced with network latency and contention issues too. Also, the central application server can become a performance bottleneck, since it is the only machine doing any work for all the users of the application.

Business logic in a distributed business tier

I wish this book included the secret that allowed us to write all our logic in one central location and avoid all of these awkward issues. Unfortunately, that's not possible with today's technology: putting the business logic in the UI (or presentation) tier, the data access tier, or the data storage tier is problematic, for all the reasons given above. But we need to do something about it, so what have we got left?

What's left is the possibility of centralizing the business logic in a business logic tier that's accessible to the UI tier, to create the most interactive user experience possible. Also, the business logic tier needs to be able to interact efficiently with the data access tier, to achieve the best performance when interacting with the database (or other data source).



Ideally, this business logic will run on the same machine as the UI code when interacting with the user, but on the same machine as the data access code when interacting with the database. (As we 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 UI developer can use to invoke any validation and manipulation logic, and it must also work efficiently with the data access tier to get data into and out of storage.

The tools for addressing this seemingly intractable set of requirements are **business objects** that encapsulate our 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 our part. The .NET Framework itself handles the details, and we can focus on the business logic and data.

By properly designing and implementing our business objects, we allow the .NET Framework to pass our objects across the network *by value*, automatically copying them from one machine to another. This means that with little effort, we can have our business logic and business data move to the machine where the UI tier is running, and then shift to the machine where the Data Access tier is running when data access is required.

At the same time, if we're running the UI tier and data access tier on the same machine, then the .NET Framework doesn't move or copy our business objects. They are used directly by both tiers with no performance cost or extra overhead. We 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 takes no extra action.

The business logic tier becomes portable, flexible, and mobile, adapting to the physical environment in which we deploy the application. Due to this, we are able to support a variety of physical n-tier architectures with one code base, where our business objects contain no extra code to support the various possible deployment scenarios. What little code we need to implement to support the movement of our objects from machine to machine will be encapsulated in our framework, leaving the business developer to focus purely on the development of business logic.

Business objects

Having decided to use business objects and to take advantage of .NET's ability to move objects around the network automatically, we need to take a little time to discuss business objects in more detail. We need to see exactly what they are, and how they can help us to centralize the business logic pertaining to our 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. `DataTables` provide an abstract and consistent mechanism by which we 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 our application's perspective, we don't need to have any understanding of how the control is rendered on the screen, or how the user interacts with it. We're just presented with 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 data and logic, but as the user of an object, we don't know *what* data or *how* the logic actually works. All we can do is interact with the object.

> **Properly designed objects encapsulate both data and any logic related to that data.**

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

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

When we create object-oriented applications, we are addressing problems of one sort or another. In the course of doing so, we use a variety of different objects. Now, while some of these will have no direct connection with the problem at hand (`DataTable` and `TextBox` objects, for example, are just abstract representations of computer concepts), there will be others that are closely related to the area or **domain** in which we're working. If the objects are related to the business for which we're developing an application, then they're business objects.

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

> **Business objects provide an abstract representation of entities or concepts that are part of our business or problem domain.**

Business objects as smart data

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

A `DataSet` or an XML document might ensure that we don't put text where a number is required, or that we don't put a number 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 to ensure that calculations or other processing is done properly against the data, without involving other objects. The data in a `DataSet`, array, or XML document is not self-aware – it's not able to apply business rules, or to handle business manipulation or processing of the data.

The data in a business object, however, is what I like to call 'smart data'. The object not only contains the data, but 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, we have much greater assurance that business rules, manipulation, calculations, and other processing will be executed consistently everywhere in our 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 Windows Forms UI, or 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.

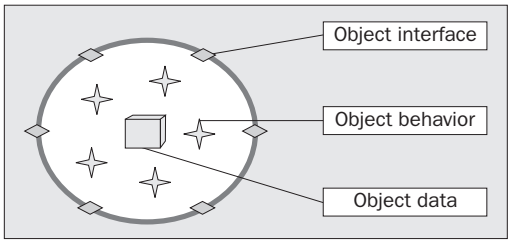
Contrast this with the `DataSet` or an XML document, where the business logic doesn't reside in the data container, but somewhere else – typically, a Windows Form or a Web Form. If this `DataSet` is used by multiple forms or pages, we have no assurance that the business logic is applied consistently. Even if we adopt a standard that says that the UI developer must invoke methods from a `Module` 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.

With 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 representing 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 our business logic scattered throughout the application, we are able to pass an `Invoice` object around. Our entire application can share not only the data, but also its associated logic. Smart data through objects can dramatically increase our ability to reuse code, and can decrease software maintenance costs.

Anatomy of a business object

Putting all of these pieces together, we 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 the following diagram:



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

Any property, method, event, or variable marked as `Public` will be available to the users of objects created from the class. For example, we might create a simple class such as:

```
Public Class Project
  Private mID As Guid = Guid.NewGuid
  Private mName As String = ""

  Public ReadOnly Property ID() As Guid
    Get
      Return mID
    End Get
  End Property

  Public Property Name() As String
    Get
      Return mName
    End Get
    Set(ByVal Value As String)
      If Len(Value) > 50 Then
```

```
          Throw New Exception("Name too long")
        End If
        mName = Value
      End Set
    End Property
  End Class
```

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

The `Property` methods, on the other hand, are `Public`. They provide a controlled access point to our object. The `ID` property is read-only, so the users of our 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.

> *None of these concepts is unique to business objects – they are common to all objects, and are central to object-oriented design and programming.*

Distributed objects

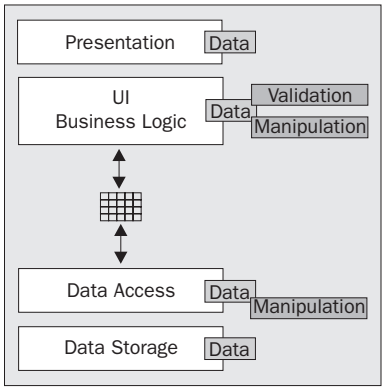
Unfortunately, directly applying the kind of object-oriented design and programming we've been talking about so far is often quite difficult in today's complex computing environments. Object-oriented programs are almost always designed with the assumption that all the objects in an application can interact 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 – worse still – on different computers.

Earlier in this chapter, we discussed various physical architectures in which different parts of our application might run on different machines. With a "high scalability intelligent client" architecture, for example, we'll have a client, an application server, and a data server. With a "high security web client" architecture, we'll have a client, a web server, an application server, and a data server. Parts of our application will run on each of these machines, interacting with each other as needed.

In these distributed architectures, we 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 it, 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 is 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 is a blend of both.

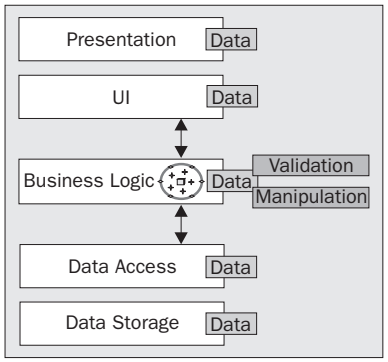
Perhaps the most common architecture for such applications is to have the data-access tier retrieve the data from the database into a `DataSet`. The `DataSet` is then returned to the client (or the web server), where we have code in our forms or pages that interacts with the `DataSet` directly.



This approach has the flaws in terms of maintenance and code reuse that we'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, speeding development.

If we do decide to stick with an object-oriented approach, we have to be quite careful: it's all too easy to compromise our OO 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 our objects and business logic. **Distributed objects** are all about sending smart data (objects) from one machine to another, rather than sending raw data and hoping that the business logic on each machine is being kept in sync.

Through its remoting, serialization, and auto-deployment technologies, the .NET Framework contains direct support for the ability to have our objects move from one machine to another. (We'll discuss these technologies in more detail in Chapter 3, and make use of them throughout the remainder of the book.) Given this ability, we can have our data access tier (running on an application server) create a business object and load it with data from the database. That business object can then be sent to the client machine (or web server), where the UI code can use the object.



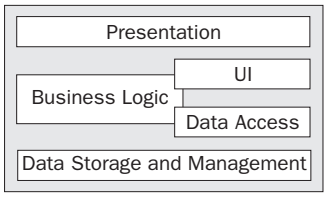
In this architecture, we're sending smart data to the client rather than raw data, so the UI code can use the same business logic as the data access code. This reduces maintenance, since we're not writing some business logic in the data access tier, and some other business logic in the UI tier. Instead, we've consolidated all of the business logic into a real, separate tier composed of business objects. These business objects will move across the network just like the `DataSet` did above, but they'll include the data *and* its related business logic – something the `DataSet` can't offer.

In addition, our business objects will move across the network more efficiently than the `DataSet`*. We'll be using a binary transfer scheme that transfers data in about 30% the size of data transferred using the* `DataSet`*. Also, our objects will contain far less metadata than the* `DataSet`*, further reducing the number of bytes transferred across the network.*

Effectively, we're sharing the business logic tier between the machine running the data access tier, and the machine running the UI tier. As long as we have support for moving data *and logic* from machine to machine, this is an ideal solution: it provides code reuse, low maintenance costs, and high performance.

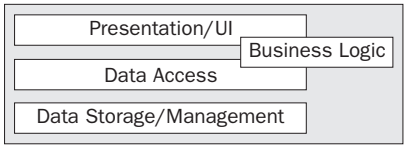
A new logical architecture

Sharing the business logic tier between the data access tier and the UI tier opens up a new way to view our logical architecture. Though the business logic tier remains a separate concept, it is directly used by and tied into both the UI and data access tiers.

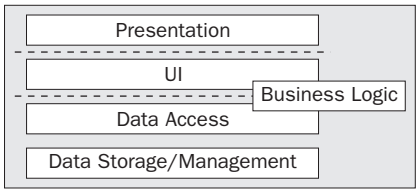


The UI tier can interact directly with the objects in the business logic tier, relying on them to perform all validation, manipulation, and other processing of the data. Likewise, the data access tier can interact with the objects as the data is retrieved or stored.

If all the tiers are running on a single machine (such as an intelligent client), then these parts will run in a single process and interact with each other with no network or cross-process overhead. In many of our other physical configurations, the business logic tier will run on both the client *and* the application server.



High scalability intelligent client



High security web client

Local, anchored, and unanchored objects

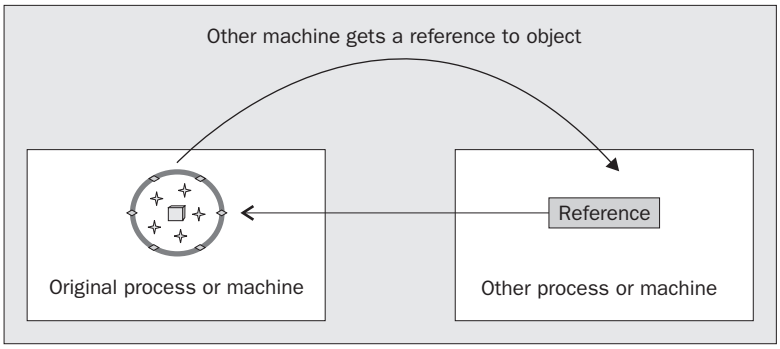
Normally, we think of objects as being part of a single application, running on a single machine in a single process. In a distributed application, we need to broaden our perspective. Some of our objects might only run in a single process on a single machine. Others may run on one machine, but be called by code running on another machine. Others still may move from machine to machine.

Local objects

By default, .NET objects are *local*. This means that ordinary objects are not accessible from outside the process in which they were created. It is not possible to pass them to another process or another machine (a procedure known as **marshaling**), either by value or by reference. Since this is the default behavior for all .NET objects, we must take extra steps to allow any of our objects to be available to another process or machine.

Anchored objects

In many technologies, including COM (Microsoft's Component Object Model), 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, while the other process or machine merely gets a pointer, or reference, back to the object.



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 must be sent across the network and processed by the object, and the results returned back across the network. This scheme is only useful if we design our object so that it can be used with very few method calls – just one is ideal! The recommended designs for MTS or COM+ objects call for a single method on the object that does all the work for precisely this reason, sacrificing 'proper' design in order to reduce latency.

This type of object is stuck, or **anchored**, on the original machine or process where it was created. An anchored object never moves – it's accessed via references. In .NET, we create an anchored object by having it inherit from `MarshalByRefObject`:

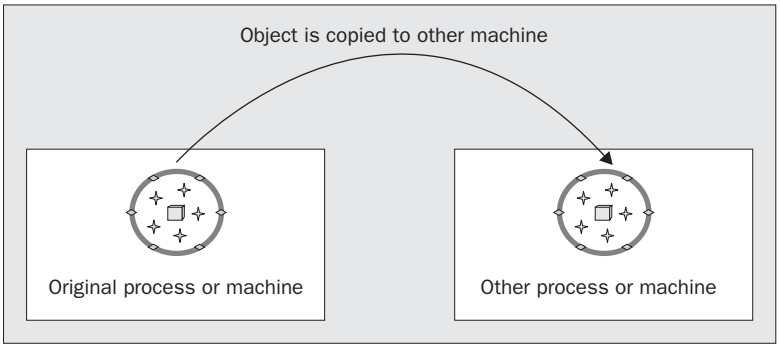
```
Public Class MyAnchoredClass
  Inherits MarshalByRefObject

End Class
```

From this point on, the .NET Framework takes care of the details. We can use remoting to pass an object of this type to another process or machine as a parameter to a method call, for example, or to return it as the result of a function.

Unanchored objects

The concept of distributed objects relies on the idea that we can pass an object 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.



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

> *One caveat here is that transferring a large object across the network can cause a performance problem. As we all know, returning a* `DataSet` *that contains a great deal of data can take a long time. This is true of all unanchored objects, including our business objects. We need to be careful in our application design to ensure that we avoid retrieval of very large data sets.*

Objects that can move from process to process or from machine to machine are **unanchored**. Examples of unanchored objects include the `DataSet`, and the business objects we'll be creating in this book. Unanchored objects are not stuck in a single place, but can move to where they are most needed. To create one in .NET, we use the `<Serializable()>` attribute, or we implement the `ISerializable` interface. We'll discuss this further in Chapter 2, but the following illustrates the start of a class that is unanchored:

```
<Serializable()> _
Public Class MyUnanchoredClass

End Class
```

Again, the .NET Framework takes care of the details, so we can simply pass an object of this type 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.

When to use which

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

Windows Forms and Web Forms objects are all local – they're inaccessible from outside the processes in which they were created. The assumption is that we don't want other applications just reaching into our programs and manipulating the UI objects.

Anchored objects are important, because we can guarantee that they will always run on a specific machine. If we write an object that interacts with a database, we'll want to ensure that the object always runs on a machine that has access to the database. Because of this, we'll typically use anchored objects on our application server.

Many of our 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 our business objects as unanchored objects, we can pass smart data from machine to machine, reusing our business logic anywhere we send the business data.

Typically, we use anchored and unanchored schemes in concert. We'll use an anchored object on the application server to ensure that we can call methods that run *on that server*. Then we'll pass unanchored objects as parameters to those methods, which will cause those objects to move from the client to the server. Some of the anchored server-side methods will return unanchored objects as results, in which case the unanchored object will move from the server back to the client.

Passing unanchored objects by reference

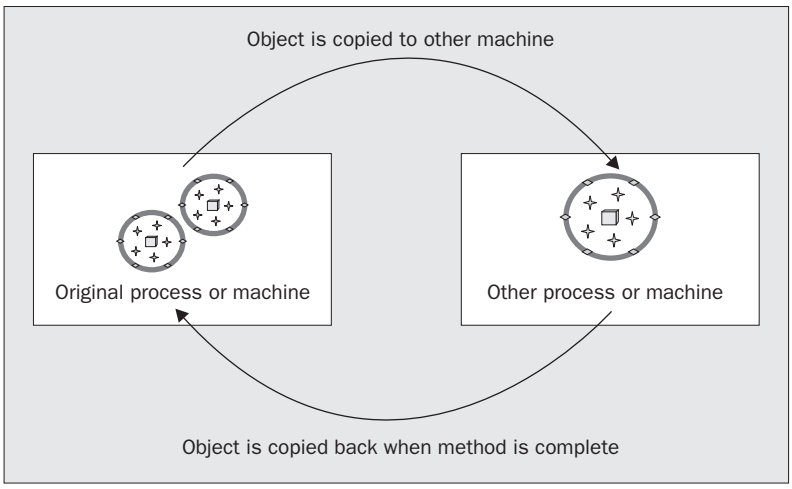
There's a piece of terminology here that can get confusing. So far, we've loosely associated anchored objects with the concept of passing by reference, and unanchored objects as being passed by value. Intuitively, this makes sense, since anchored objects provide a reference, while unanchored objects 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 were not reflected in the original value. That makes sense, since the changes were made to a copy, not to the original value.

In distributed applications, things get a little more complicated, but the definitions above remain true: we can pass an object by reference, so that all machines have a reference to the same object on some server; and we can pass an object by value, so that a copy of the object is made. So far, so good. However, what happens if we mark an object as `<Serializable()>` (that is, mark it as an unanchored object), and then *intentionally* pass it by reference? It turns out that the object is passed by value, but the .NET Framework attempts to give us the illusion that it 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 back to the calling code when the method is complete, and our reference to the original object is replaced with a reference to this new version.



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

> **If we pass an unanchored object by reference, we must always make sure to update *all* our references to use the new version of the object when the method call is complete.**

We can choose to pass an unanchored object by value, in which case it is passed one way, from the caller to the method. Or we can choose to pass an unanchored object by reference, in which case it is passed two ways, from the caller to the method, and from the method back to the caller. If we want to get back any changes the method makes to the object, we'll use by reference. If we don't care about, or don't want, any changes made to the object by the method, then we'll use by value.

Note that passing an unanchored 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

Hopefully, at this point, your imagination is engaged by the potential of distributed objects. The flexibility of being able to choose between local, anchored, and unanchored objects is very powerful, and opens up new architectural approaches that were difficult to implement using older technologies such as COM.

We've already discussed the idea of sharing the business logic tier across machines, and it's probably obvious that the concept of unanchored objects is exactly what we need to implement such a shared tier. But what does this all mean for the *design* of our tiers? In particular, given a set of unanchored or distributed objects in the business tier, what's the impact on the UI and data access tiers with which the objects interact?

Impact on the UI tier

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

More importantly, since the business objects are distributed (unanchored), they'll end up running in the same process as the UI code. Any property or method calls from the UI code to the business object will occur locally without network latency, marshaling, or any other performance overhead.

Impact on the data access tier

The impact on the data access tier is more profound. A traditional data access tier 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. If we now encapsulate the data access logic inside each object, what is left in the data access tier itself?

The answer is twofold. First, for reasons we'll examine in a moment, we need an anchored object on the server. Second, we'll eventually want to utilize Enterprise Services (COM+), and we need code in the data access tier to manage interaction with Enterprise Services.

> *We'll discuss Enterprise Services in more detail in Chapters 2 and 3, and we'll use them throughout the rest of the book.*

Our business objects, of course, will be unanchored, so that they can move freely between the machine running the UI code and the machine running the data access code. However, if the data access code is inside the business object, we'll need to figure out some way to get the object to move to the application server any time we need to interact with the database.

This is where the data access tier comes into play. If we implement it using an anchored object, it will be *guaranteed* to run on our application server. Then we can call methods on the data access tier, passing our business object as a parameter. Since the business object is unanchored, it will physically move to the machine where the data access tier is running. This gives us a way to get the business object to the right machine before the object attempts to interact with the database.

> *We'll discuss this in much more detail in Chapter 2, when we lay out the specific design of the distributed object framework, and in Chapter 5, as we build the data access tier itself.*

Architectures and frameworks

Our discussion so far has focused mainly on architectures: logical architectures that define the separation of roles in our application, and physical architectures that define the locations where our logical tiers will run in various configurations. We've also discussed the use of object-oriented design and the concepts behind distributed objects.

While all of these are important and must be thought through in detail, we really don't want to have to go through this process every time we need to build an application. It would be preferable by far to have our architecture and design solidified into reusable code that we could use to build all our applications. What we want is an application **framework**.

A framework codifies our architecture and design to promote reuse and increase productivity.

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

All of that architectural discussion, decision-making, design, 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 for us to create actual business applications. It's an overhead that in the long term we 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 of our 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 reverse! We typically spend far too little time analyzing our business requirements and developing good application designs to meet those business needs. Part of the reason is that we often end up spending substantial amounts of time analyzing and designing the 'plumbing' that supports the business application, and we run out of time to analyze the business issues themselves.

What I'm proposing here is that we can reduce the time spent analyzing and designing the low-level plumbing by creating a framework that can be used across many of our business applications. Is the framework that we'll create 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, we've focused on the theory behind distributed systems – specifically, those based on distributed objects. The key to success in designing a distributed system is to keep clear the distinction between a logical and a physical architecture.

Logical architectures exist to define the separation between the different types of code in our application. The goal of a good logical architecture is to make our code more maintainable, understandable, and reusable. Our logical architecture must also define enough tiers to enable any physical architectures that we may require.

A physical architecture defines the machines on which our application will run. An application with several logical tiers can still run on a single machine. That same logical architecture might also be configured 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 our specific environment.

The tradeoffs in a physical architecture for an intelligent client application are very different from those for a web application. A Windows application will typically trade performance against scalability, while a web application will typically trade performance against security.

In this book, we'll be using a 5-tier logical architecture consisting of presentation, UI, business logic, data access, and data storage. We'll be using this architecture to create Windows, web and Web services applications, each with a different physical architecture. In the next chapter, we'll start to design the framework that will make this possible.