Chapter 9
Three-Tiered Development

Lately there's been a lot of hype about three-tiered development. The main idea is to split applications into different layers—the interface, the business logic and the data back end. Many people tout three-tiered development as the best thing since sliced bread. While it can't cure world hunger, it definitely helps me a great deal in my development efforts.

A brief introduction

Every application has some kind of user interface, attached business logic and data back end. Of course, each part can have a different appearance. The interface might be a regular windows interface, a Web page, a voice interface or even an automated process or script. The data might be stored in Visual FoxPro tables, in SQL Server databases, in XML files or any other kind of storage. The business logic might be compiled into an EXE or some kind of COM component. None of these pieces are specific to three-tiered architecture. You'll find them in any kind of business application, whether it be a modern component-based Windows application or an old mainframe monster.

A modern software development approach has to be a lot more flexible and powerful than in past days. Changes must be implemented more quickly, cheaply and at a higher quality. Applications are no longer monolithic. Data sources such as ADO, SQL Server and XML must be used in addition to regular Visual FoxPro data. Different interfaces must be used, including HTML, Windows interfaces and more exotic ones (at least for Visual FoxPro developers) such as Windows CE. Even if you can do everything using FoxPro data and a regular Windows interface, your app might need to talk to other components such as Microsoft Office. On top of that, you will definitely need flexibility so you can quickly change parts of an interface or the business logic without worrying about overall project quality or going through an enormous amount of work.

The three-tiered model is designed to reduce design complexities and increase the application developer's flexibility in creating, maintaining, and redeploying applications on different platforms.

In the three-tiered approach, the interface, business logic and database layers are kept separate. This can be done in various ways. All components are compiled separately in COM components and EXEs, or the classes are simply kept in different libraries and inheritance trees, but they are still compiled into one EXE, possibly using a monolithic Visual FoxPro approach. The key is that the layers talk to each other *only* through a defined API and are *not* linked in any other way. This means that, at any time, you can modify, change, or even replace a tier, and—assuming that you continue to write to the same inter-tier API—keep the entire system working. The independence that you gain is the key to this approach. The rest of this chapter is dedicated to explaining, supporting, defending, and preaching about this idea.

The model-view-controller approach

Like many things in object-oriented development, the three-tiered architecture isn't really new. Many of the traditional object-oriented languages such as Smalltalk have been using this approach for quite a while. It was (and is) known as the "model-view-controller" design, where the "model" is the data, the "view" is the interface and the "controller" is the business logic. The Unified Modeling Language (UML—see Chapter 12) replaced this terminology with the terms "user services," "business services" and "data services."

Better than sliced bread: sliced applications

Conceptually, the idea of three-tiered development is simple: You take your regular applications and slice them in three layers. Interface elements belong in the interface layer. Business rules, logic, and data validation belong in the middle tier, and the data store is in the data layer. While this sounds easy, it's a significant change of view for the "traditional" FoxPro developer. Because we have always had data controls that are "intimate" with the data, we have always been able to design interfaces that directly access data in our tables and, with the VALID clause, do validation in the interface. The "one-tier" model (or one and a half if you want to argue about editing memory variables instead of the table) was the natural way to go and, with FoxPro, worked quite well.

Well, Visual FoxPro is now playing in a larger arena and, if you want to structure applications to play the new game, you are going to have to move away from the old ways of doing things. You cannot put your logic into the Valid event of a field in a form. Neither can you directly bind fields to data. You can't even have these parts in the same composite object. Fortunately, all this complies with most of the principles of proper object-oriented design. Rather than using a part of the interface to validate entries, you would call out to a behavioral object. Data would be bound to the interface through well-defined object interfaces. The actual implementation can vary greatly, depending on whether you create COM components or a monolithic Visual FoxPro application (see below).

The most difficult part of three-tiered development is following the rule of keeping the three tiers separate. The interface and business logic components are especially hard to separate. It's tempting to add some code to one of the controls in a form rather than creating another method in a behavioral object that does all the calculations. Unfortunately, as soon as you take such a shortcut, you break your entire three-tiered model. This will set you back to the prior level of software development. Self-control (if you are developing an application by yourself) and code reviews (in team development scenarios) are key. In most areas of development, you can get away with bending the rules every now and then. This is not true for three-tiered architecture. Bend the rules once and you'll lose all the advantages but still carry all the burdens.

Three-tiered internally

When you hear the term "three-tiered", you might immediately think of an application that is compiled in various COM components that talk to some back-end server (possibly on a network) to retrieve data and a number of different interfaces to interact with the user. COM, ADO (or ODBC) and MTS (Microsoft Transaction Server) are the key technologies that make these scenarios work. Most of these things seem strange and unnatural to Visual FoxPro developers who are used to a straightforward way of making things happen.

However, this is only one of the possible scenarios. Another approach is to stick to Visual FoxPro (or any other environment) and compile all the tiers into one EXE. The main advantage of three-tiered development is the flexibility and ease of maintenance you gain. The fact that you have to recompile your application in order to switch components, interfaces or data sources isn't usually a big problem. After all, compiling takes only a couple of minutes, even for large projects. The fact that you can change interfaces or data sources in a matter of minutes, on the other hand, weighs heavily.

I often reuse a certain framework for my consulting customers. This framework follows a strict three-tiered approach. When I initially designed the framework, nobody was interested in this kind of architecture (not the FoxPro or Visual Studio world, anyway). Important technologies such as ADO and MTS weren't even planned at that time. For this reason, I designed my three-tiered application using only Visual FoxPro technology. In other words, this application was a monolithic Visual FoxPro application, yet it was strictly three-tiered.

Over time, as new technologies emerged, I enhanced my framework. Now I can use it to create COM components that are called from Visual Basic or Active Server Pages as well. However, I often use the old approach simply because the majority of applications still run in a regular Windows network environment and scalability is not a major issue. (Visual FoxPro still is pretty fast at handling data (whether it is FoxPro tables or SQL Server databases.)

Let me introduce some basic ideas behind my framework.

One of the main design goals was to use different data sources without changing any of the business logic or interface components. Another requirement was to use different interfaces (at this point, mainly Windows and plain HTML interfaces). On top of that, I wanted to be able to switch the business logic layer, mainly to make sure I could handle multi-lingual and (more importantly) multi-cultural issues as well as adjustments to serve different branches of the targeted businesses. This requirement was relatively trivial, yet most three-tiered applications don't handle that very well. Usually only the interface and the data source can be switched, but the logic remains the same. (I guess by now you get the idea that I don't particularly like this approach.)

Handling the data

In order to handle the data generically, I use controller objects. (This term wasn't chosen very wisely. "Model" or "DataService" would have been more appropriate.) An abstract controller defines the object interfaces, and there are subclasses for each of the data back ends I want to talk to. Originally, the framework was designed to handle Visual FoxPro and SQL Server data. Now it handles Oracle as well as ADO.

To get to data, you can use the controller's query methods, which can create regular Visual FoxPro cursors as well as objectified data (see below). The way I talk to the controllers never changes. The controller serves as a translator between my attempts to retrieve data and the language spoken by each specific back end.

It might surprise you that the controller objects are part of the back end (data layer). Typically, the data back end simply is a collection of data in a standardized format such as SQL Server, FoxPro tables or XML. However, there is no reason why you couldn't create objects that belong to the data layer. Many of today's products such as SQL Server and ADO represent the object part of the data layer. My controller objects simply add another layer of abstraction to this scenario, thus making it more generic.

When Microsoft first released ADO, I was concerned that the additional layer I built would be redundant, but this concern proved wrong. Today, I still talk to SQL Server directly using SQL Pass Through (mainly for performance reasons) and not ADO. I also use XML data sources directly (using the ActiveX control provided by Microsoft). And what if I use plain Visual FoxPro data? Should I retrieve that through ADO? I don't think so! So far I have been satisfied with this additional layer, and I would redesign it in the same manner without hesitation.

Creating the interface

In most three-tiered applications, the interface is the driving force that invokes business logic, which then retrieves the data. However, this is limiting because the interface decides what kind of business logic to invoke, which automatically defines what data to use. This would be fine in scenarios where my main concern is reusing components in different interfaces or applications, but as mentioned above, not only do I want to reuse middle-tier components (business logic), I also want to be able to exchange these components in a flexible manner. If I were to use the interface to invoke those objects, I would need to change every interface after I introduced new middle-tier classes.

This scenario didn't work for me, so I created special objects that are responsible for launching the interface. These objects are my "UserService" objects. Again, I have an abstract user service object that I subclass into a user service object for every interface I want to support. Initially, the interface would be either a regular Visual FoxPro Windows interface or an HTML-based approach. By now I've enhanced this so any kind of COM component can require interface operations.

The user service object provides a number of standard operations, such as loading some data for editing (single items) or displaying a list. Any user service object can be decorated, so it talks to a controller to retrieve the correct data. Depending on whether I'm using a regular Windows interface or another component, the user service either launches a form or creates HTML that will eventually travel across the wire. Launching a form is trivial. Creating HTML pages is somewhat more complex. Basically, the HTML user service object requests data, merges it into HTML templates and sends it out. Initially, the system was designed to work with the West Wind WebConnection (www.west-wind.com) and Visual WebBuilder (www.visual-webbuilder.com). Now I've enhanced it so it can serve as a COM component that's called from Active Server Pages or any kind of other COM client that can handle HTML. Once the user modifies the data, the request hits the Web server again and the user service object gets involved. The user service collects all the data in the page, reassembles regular Visual FoxPro data, and hands it back to the rest of the application—which doesn't even know what kind of interface was utilized.

Both the regular VFP user service and the HTML user service actively create a user interface. In the case of an HTML interface, additional rules are attached in order to reduce traffic. In the case of a Visual FoxPro user service, the interface directly calls the business logic layer to validate and handle data. The HTML user service does this as well, but only when the user submits data. This is the final and most important data validation step. All validation that is done right in the Web page (using scripts) is very basic and doesn't cover complex business rules. That's fine. The main purpose here is to eliminate stupid problems such as submitting an empty form. Whether the data that has been submitted actually makes sense is hardly ever validated in the page itself. This helps to reduce the number of hits and total traffic.

In the case of the COM user service, things work slightly differently. This user service doesn't create an interface, but it does create a composite object that contains all data to be used in the interface, and it has some very basic business rules that are implemented through access and assign methods. This object is then sent through COM channels, and it's the responsibility of the client to create the actual interface. This way, I can use any COM client (such as Visual Basic) to provide the interface.

The user service objects are configured at compile time. I have a couple of compiler directives (#DEFINE) that specify what kind of user service object I want to use. The user service objects don't get to decide what kind of controller or logic object will get involved. To load customer data, for instance, the user service object would simply invoke the customer controller. Whether this controller is subclassed from the VFP controller class, the ADO controller class, or any other controller, is defined elsewhere (see below). The same is true for the business logic. The user interface would simply invoke a "tax-calculation object." The class this object is made of depends on a number of settings, such as the country the application is used in, or the country/state you are dealing with. These things can be configured at compile time as well as during runtime, depending on the kind of business logic you need to invoke (see below).

Invoking the business logic

Creating the business logic layer isn't quite as straightforward as creating the other two layers. The business logic layer is responsible for getting data from the data layer, presenting it to the interface, receiving edits, validating them against business rules, and then sending the results back to the data layer.

Creating abstract parent classes is difficult because you'll encounter various different needs. You could create an abstract logic base class that had a number of standard methods, but you would soon discover that those methods would hardly ever match your needs. And that's fine. After all, the business logic is what programming is all about. Our target must be to reduce the effort it takes to resolve technical issues, but the business logic often will be coded individually.

However, polymorphism will be important within certain kinds of business logic objects. You should create an abstract class for all your tax-calculation objects, for instance. This will allow you to exchange different objects without changing the rest of your system. Another typical example would be an object that validates whether addresses were entered correctly. Depending on the country you are in, different rules will apply, so you should create different classes for each country, all subclassed of one abstract parent class to keep the interface persistent. However, it isn't that important for the tax-calculation and the address-verification objects to share the same interface. What are the chances you will rip out the "U.S. tax-calculation" object from your invoicing module and replace it with the "European-address-verification" object? Not very high, I would say, unless you want to check whether the invoice total coincidentally is a valid ZIP code, or something like that.

This leaves us with the dilemma of not having a clear approach to invoking the business logic. For this reason I decided to introduce yet another set of abstract classes that are used to create instances of business logic. All they do is return object references to the business logic object that's appropriate for the current use. There would be one of those objects for each of the logic objects I have. For instance, a tax-calculation business service object would have a GetHandle() method that retrieves or creates a reference to a business logic object and returns it. From this point on, I would directly talk to the business logic object rather than the business service object.

The way the business service object decides what logic object to invoke varies greatly. In the tax-calculation scenario, many decisions might be made at runtime. Depending on where the customer is located, different objects will be invoked. However, there might also be some configurations that happen at compile time. When I create a U.S. version, an entirely different set of logic objects will be compiled into the product than when I create a European version. After all, when shipping something from the U.S. to Germany, the tax will be calculated differently than when shipping from Austria to Germany, even though the destination country is the same.

Compiling one EXE

By now you know the ideas behind the three tiers, but you have yet to explore how the entire application is compiled. In many scenarios, the user interface is the part that contains or invokes the rest of the application. As mentioned above, I don't like this approach. I like to use an object that works as a launch pad and coordinator for all other tiers. This is my application object. It asks the user service object to provide a starting point (the main window, or the home page) and it is used to define application-global settings such as what objects are to be utilized. This removes a lot of responsibility from the interface layer. Note that this object doesn't have a lot of code. It would not handle the instantiation of interface objects, for example, but it would have a property (or something similar) that would tell us whether the current interface is Windows-based or Web-based.

Displaying and manipulating data

One of the most difficult parts of three-tiered development is transferring data from the back end to the interface. It's easy to run a query in the data service object (controller), but how do you get that cursor into a form's data session? Well, there are a number of different approaches. I like to use objectified data. In other words, I create a data object representing a record (or many data objects representing a record set). The objects have only properties, and each property represents a field in a table. The objects are created by the controller and handed over to all kinds of interface components. The user service object is responsible for handing the object to a form, merging the object into an HTML template, or creating a composite object if a COM client makes a request. This approach works fine for single records or small record sets (up to a couple of hundred records). However, it doesn't work very well for large data sets. In this case, performance won't be all that great, and resources will run out quickly. Also, Visual FoxPro grids cannot use these kinds of record sets as the data source.

Another approach is to use the controller objects to create regular Visual FoxPro cursors in the data session of a certain form (or other interface component). To do so, the controller has to switch data sessions before a cursor is created and before data has to be saved. In this case, you need to be very careful resetting the data session. If the controller fails to restore initial settings, the entire application is likely to get confused and to malfunction. Data objects are a much safer approach, so I try to stay away from the "session hopper" scenario wherever possible.

As a general tip, I recommend not using grids for data entry. Interfaces that use grids are very hard to implement in other interfaces such as HTML. I use grids almost exclusively for display purposes.

Class and inheritance hierarchies

You've probably heard it a hundred times by now: "Never use the Visual FoxPro base classes! Create your own set of classes, put them in a library called Base.vcx and base all your other classes and controls on this set."

This is still true in three-tiered applications, but you shouldn't use this approach in all tiers. If you want to stick to this approach, keep separate sets of base classes for each tier. In other words, create the libraries "user service base.vcx", "data service base.vcx" and "business logic base.vcx". Make sure you never base classes in different tiers on base classes belonging to another tier. This will tie the tiers together, which moves you closer to single-tiered development again.

Obviously not all base classes are required in each tier. Most of Visual FoxPro's base classes are interface related. You don't need those classes anywhere but in the "user service base.vcx". Typically, the middle-tier and the back-end classes are all based on *Custom*. Sometimes you might see classes such as "line" or "separator," because those classes are resource-friendly (unlike the heavy *Custom* class). However, those classes never become visible.

In Visual FoxPro, every class must be based on a Visual FoxPro base class. You cannot start from scratch. However, this is what most people mean to do when creating middle-tier or back-end objects. So let's just assume for a minute that we could start out with a brand-new class. We would create abstract classes, create concrete subclasses, and so forth. We would have many different inheritance trees starting from scratch. We would have a tax-calculation tree, we would have an address-verification tree, and so forth. What do those trees have in common? Nothing! They all are independent classes. Now let's go back to the real Visual FoxPro world, where we can't start from scratch but have to base everything on the *Custom* base class instead. What would those trees have in common, other than the fact that Visual FoxPro forced them to use an unnecessary parent class? Not a bit more than the classes in the example above! So does it make sense to create a set of base classes for the logic layer and the back end? I doubt it. In fact, it will make it harder to reuse those components because they always rely on some parent classes that you have to drag over into other projects. These projects might use their own set of base classes for the middle and back-end tiers. You can now redefine your classes (making it impossible to reuse updated versions of that class) so they use the new set of base classes, or you can maintain multiple sets of base classes, which would defeat the purpose of the concept altogether. And then again, what would be the benefit in this situation? I don't know. So I recommend creating your own set of base classes for the interface tier only.

Exposed tiers

Let's return to the scenario of three-tiered development using COM components. In this case, you would design your application in a similar fashion as described above. In some regards, it's even simpler to create COM components than an EXE. You aren't in danger of breaking the rules of three-tiered development, because you simply can't. Reusing components is much easier because you don't have to worry about dependencies. Creating clean class hierarchies is also easier because you're dealing with various projects, which reduces the risk of creating unwanted relationships. Creating good object interfaces comes quite naturally, because there is no way to talk to a COM component other than through its interface. You aren't in danger of cheating (which might happen unknowingly, but with the same complications nevertheless).

Unfortunately, a number of issues are quite a bit harder to resolve in the world of COM components than in a regular Visual FoxPro application. How do you display the data, for instance? Again, single records aren't a problem, because you can always create a data object, but there is no way to switch into some other component's data session. That's where ADO comes in. ADO handles all data as objects, which can be passed through COM channels from the back end through the middle tier into the interface.

The great advantage of COM-based three-tiered applications is language independence. You can create a data-specific component in Visual FoxPro, another one that does a lot of calculations in Visual C++, a third one in Visual J++ and yet another one in Visual Basic. Even though you might not want to do that right away, you always have the option to switch to different components later on. If you switch to Visual FoxPro from Visual Basic, for instance, and aren't quite familiar with the environment, you can create a performance-critical data-retrieval component in Visual FoxPro, and other components that aren't so critical in Visual Basic (the tool you're familiar with). Later, when the application grows and performance becomes critical in other parts, you can easily trade single components for faster ones.

COM-based, three-tier applications often run on servers, whether Web servers or just intranet servers. In these scenarios, one powerful computer does almost all the work. When you have a large number of users, a large number of hits, or both, you need to make sure your application scales well. This isn't a trivial task in monolithic Visual FoxPro applications. Visual FoxPro is fast, no doubt, but once a certain amount of traffic occurs, there isn't much you can do. There's no intrinsic multithreading, no load balancing, basically nothing you can do other than coding your own pool manager (good luck!). When your application is based on COM components, this becomes much easier. You can simply let Microsoft Transaction Server handle your components, and it will make sure enough resources are available. All you need to do is register your component, and MTS will manage all calls to it. This includes component instantiation, which is handled entirely by MTS to ensure there are enough resources and component instances. Essentially MTS is a very advanced pool manager.

However, there is more to MTS than just scalability. "Transaction" is its middle name, which indicates that MTS is a transaction manager system as well. In fact, it allows you to span transactions over multiple components, no matter what language they are written in. In this case, things don't happen automatically anymore, but you have to design your COM components specifically for MTS. This isn't rocket science, but it is beyond the scope of this book. If you are interested in MTS, visit my Web site (www.eps-software.com) where I keep a list of recommended reading and a number of MTS articles.

Conclusion

Three-tiered development isn't as difficult as many people would have you believe. The easiest way to get started is to create a small prototype that handles some dummy data. Try to keep the data in different databases and create different interfaces, following the rules outlined in this chapter. Once you are more comfortable with this approach, you can try to create different logic components that can be swapped (static at first, and dynamic later on). You will soon realize the advantages of this approach, and I promise you, once you get used to the slightly different way of doing things (and the handful of complications), you won't want to go back to the old way. In fact, single-tiered applications now seem like source-code chaos to me. Three-tiered design helped me to raise the quality of my code tremendously—not only because of the obvious advantages I laid out in this chapter. Many design questions, such as what classes should be based on your own set of base classes, will become easy to answer once you take this well-organized approach.