Chapter 2
Visual FoxPro for Client/Server Development

After reading Chapter 1, you should have a good understanding of what makes client/server databases different and why you might want to use them. But the $64,000 question is: Why would you use Visual FoxPro to develop your client/server applications? This chapter will answer that question for you, the developer, and provide answers that you can give if your clients ask this question.

More than half of all software developers today use a certain variation of Beginner's All-purpose Symbolic Instruction Code (BASIC), according to Microsoft. Why? Partly because of familiarity, partly because of its versatility, and also partly because popularity begets popularity. But you might recall your parents telling you, "Just because everyone else does it doesn't mean you should do it, too." This chapter discusses six features that make Visual FoxPro the best client/server rapid application development tool available today: object-oriented programming, COM support, built-in client/server support, built-in local data engine, support for other data-access technologies such as ADO, and Rapid Application Development (RAD). Many of these features could be, or are, the topics of one or more books of their own. While each feature is discussed separately, keep in mind that it is the combination of these features that makes VFP such a great client/server development tool. Not many development tools can offer the combination of features found in FoxPro. Visual Basic, for example, offers great COM support and support for ADO and is an excellent RAD environment, but it isn't object-oriented, while C++ is a great object-oriented programming language but lacks built-in client/server or local database support and has never been accused of being good for RAD.

Object-oriented programming (OOP)

The holy grail of software development has long been code reuse. In a procedural-programming paradigm, we can achieve a certain level of code reuse by writing libraries of functions/procedures that can be called as needed. Object-oriented programming, or *OOP*, makes code more reusable through the use of *objects*. In the real world, we are surrounded by objects, each of which has physical characteristics and many of which can do certain things. In object-oriented programming terminology, an object's characteristics are called *properties*, and the things an object can do are called *methods*. A person has certain characteristics, such as name, address, gender, height, weight, bank balance and so forth. A person can also perform actions, such as writing down all of his or her characteristics. A programming object that represents a person would have properties for each of his or her characteristics. The person object can also have a method that would write down—or print out, if you will—these properties. Representing real-world entities with programming objects in this way, while an integral part of object-oriented programming, is called *object-based programming*. Most

popular, modern programming languages such as Visual FoxPro, Visual Basic, C++ and Java support object-based programming. Object-based programming, through its use of abstract objects to represent real-world objects, is a step in the right direction for improving code reuse.

There are two features common to all object-based programming languages: *encapsulation* and *polymorphism*.

Encapsulation is the combination of data and code into a single entity. The data, in the form of memory variables belonging to the object, are called properties. The code, called methods, gives the object its ability to manipulate its data and to perform other actions. To represent the first name of a person in a procedural language might require a global memory variable called gcFirstName. The value of the variable would be set like this:

```
gcFirstName = "Kim"
```

An object would encapsulate the first name in a property as part of the object:

```
oPerson.FirstName = "Kim"
```

Rather than trying to keep track of numerous memvars for each of numerous persons, objects require that the programmer maintain memvars only for each object.

The person object can also contain code such as the ability to print out its properties. This code is known as a method and might be called Print(). To print the characteristics of a person in a procedural program, you might have a function called PrintPerson() to which you would pass a parameter for each of that person's characteristics:

```
PrintPerson(gcFirstName, gcLastName, gcMiddleInitial, gcHeight, etc.)
```

While such a function is certainly reusable, it isn't reused easily. An object, on the other hand, could have a Print() method that contains all the code necessary to print its properties. It could be called like this:

```
oPerson.Print()
```

Which call would you rather make over and over again? More importantly, which call, when made over and over again, is likely to contain fewer errors and require less debugging?

Polymorphism is the ability of multiple objects to share the same interface. Having the same interface means that the properties have the same names and data types and the methods have the same names, parameters and return types. In the real world, it is clear to everyone that programmers and salesmen are not the same. But despite that, they have the same interface, as both programmers and salesmen have names, addresses, height, weight and so on. Furthermore, all programmers and most salesmen can write down their characteristics, though they might do it differently. A salesman, for instance, would undoubtedly write a much longer description than a programmer. So the code in a salesman object's Print() method would be different from the code in a programmer's Print() method, but when it comes to using the objects, they are both manipulated in the same way.

Object-oriented programming goes one step further. Just as certain real-world entities, such as children, can inherit the characteristics and abilities of their parents, so too can

programming objects in an object-oriented language inherit properties and methods from other objects. A programming language, to be object-oriented, must support not only encapsulation and polymorphism, but also *inheritance*. In an object-based language, a programmer object and a salesman object would each require their own code. But in an object-oriented language, they could share as much code as appropriate for their common functionality. Since each is a person, you could create a person object with all the properties and methods appropriate for all persons. But since they require different Print() methods, you would then create a programmer object and a salesman object, each of which inherits all the characteristics of a person object. Then you'd write a programmer.Print() method that prints concisely and a salesman.Print() method that is long-winded. In pseudo-code, it might look something like this:

```
DEFINE CLASS person AS object
DEFINE CLASS programmer AS person
DEFINE CLASS salesman AS person
```

While object-based programming certainly enhances code reuse by simplifying the way the code is used, object-oriented programming allows a quantum leap in code reuse because of inheritance. Of the four languages in Microsoft's Visual Studio suite—Visual FoxPro, Visual C++, Visual J++ and Visual Basic—all but Visual Basic are object-oriented. Visual Basic is "object-based" because it does not support inheritance.

There are two different types of inheritance: single inheritance and multiple inheritance. Single inheritance means that a child object can inherit the properties and methods of a single parent, while multiple inheritance means that a child can inherit from multiple parents. With multiple inheritance, an object normally inherits all the properties and all the methods from a parent. If an object were created that inherited from both a programmer and a salesman, it would have a concise Print() method *and* a long-winded one. While multiple inheritance offers versatility, it is also more difficult to manage than single inheritance. C++ fully supports multiple inheritance. To simplify the management of multiple inheritance, some languages, such as Java, support single class inheritance and multiple interface inheritance. Visual FoxPro supports single inheritance.

Support for COM

Object-oriented programming is a huge boon to code reuse. However, OOP is a language-centric solution. To take advantage of OOP, not only must all your objects be written in the same programming language, but in most cases you must have the source code. This is not a problem if the objects are written within your company, but wouldn't it be nice to take advantage of objects written by others as well? And while we're at it, wouldn't it also be nice if you could place objects on different machines in an organization in order to spread out resource utilization, simplify distribution or enhance security? Object orientation doesn't help in either of these cases. What is needed is an object-brokering technology such as Microsoft's Component Object Model, or *COM*. COM is the key to building applications that can communicate with one another, take advantage of components or applications from other vendors, or be distributed among different computers on a network.

COM, the technology, is used to provide four different features:

- *ActiveX documents* allow a user to work in one application while manipulating a document using another application.
- *ActiveX controls* allow developers to include additional functionality, typically in the form of a user interface, within their applications.
- *Automation* allows one application to control or communicate with another application or component.
- *Remote Automation*, or *Distributed COM (DCOM)* allows components to reside on different computers, a strategy known as distributed applications.

Visual FoxPro supports all of these different flavors of COM.

Visual FoxPro can be a client for *ActiveX documents*. With this technology, a VFP application can allow a user to manipulate linked or embedded documents (ActiveX documents were once called OLE, or Object Linking and Embedding, documents), such as a Word document or Excel spreadsheet, using the menu and toolbars of Word or Excel, but without leaving the VFP application or starting Word or Excel. Visual FoxPro can also act as a server for ActiveX documents, allowing a VFP application to be hosted by another application like Word or Excel.

Visual FoxPro applications can use *ActiveX controls* to provide users with functionality that is difficult or impossible to implement using VFP's native controls, or to take advantage of work already done by others. A great example of this is the Internet Explorer control. With a huge investment, you could probably figure out how to create a Web browsing form in Visual FoxPro. But by simply dropping the free IE Browser ActiveX control on a VFP form, you can include Web browsing and HTML viewing capabilities in an application without that huge investment.

While most ActiveX controls provide some sort of user interface, not all do. For several years, there has been a Crystal Reports ActiveX control that allows developers to print, view and manipulate reports within an application. That control has no user-interface functionality at all. Seagate is currently replacing it with an Automation component.

While Visual FoxPro can use ActiveX controls, it cannot create them. You must do so with other tools such as Visual C++ or Visual Basic.

Automation allows a client application to communicate with or control a server component or application using COM. When Automation support was first introduced in VFP in version 3.0, VFP could be used as a client to control another server application, such as Word or Excel. As an example, consider an application written in 1995 that public health officials could use to keep track of certain health issues in Third World countries. Because of budget limitations, it was not affordable to write a module to attempt to find best-case investment strategies that would reduce their national burden of disease. Instead, the solution was to use Automation to control Excel's Solver component to try to find the best strategy. The total time invested was less than one day!

Since version 5.0, Visual FoxPro has also had the ability to create server components. Now, in addition to being able to control other applications, other applications can control your application or other components you create. Furthermore, multiple applications you or your company write can make use of components you write. For example, suppose you had a need for a component that allowed import/export of data in a national standard ASCII format, and

two places within your application needed to use this component, and other applications needed access to it, too. You could create a COM component to encapsulate that functionality. Since it was built as a COM component, any application, not just your VFP application, could access the methods that perform the import and export.

The final piece in the COM puzzle is the ability to distribute Automation components on multiple computers. The first iteration of this was called *Remote Automation*, but it has mostly been supplanted by Distributed COM, or *DCOM*. Why distribute components on different computers? For the same reasons you separate the application from the database in client/server computing. In fact, we consider distributed applications to be merely another step beyond client/server. You separate client applications from server databases for performance, scalability, security and cost-effectiveness. All the same reasons apply to distributing components among different network resources.

Regardless of where Automation servers reside—on the local computer or another computer on the network—their use in a client/server application turns at least part of the application from a two-tier design into three-tier one. You might create a three-tier application using stand-alone components running remotely, or you might create components that can run in some other Automation host environment such as Microsoft Internet Information Server (IIS) or Microsoft Transaction Server (MTS). Both IIS and MTS are multi-threaded hosts for improved scalability. Visual FoxPro allows you to create multi-threaded COM components that scale well in either host, as well as any others that support apartment-model threading.

Built-in client/server support

Visual FoxPro has support for client/server databases built right in. In fact, VFP supports not just client/server databases, but any ODBC database. It can even connect to VFP data via ODBC rather than natively, as discussed in Chapter 7, “Downsizing.” Many other popular client/server development languages, such as Visual Basic or C++, have no built-in support for data of any kind and require either calling a database server’s API or using data-access libraries or components. Database applications written with Visual Basic, for example, would provide data access with DAO (Data Access Objects), RDO (Remote Data Objects) or ADO (ActiveX Data Objects), depending on what year it was written and what database was in use.

VFP uses ODBC (Open Database Connectivity) to connect to client/server data. ODBC is currently the most widely used database connectivity technology. Visual FoxPro lets you use its ODBC features either with remote views, which are pre-defined, updatable queries, or with SQL pass through, which allows you to send any supported command to the database server. Remote views are covered in Chapter 4, “Remote Views,” and SQL pass through is covered in Chapter 6, “Extending Remote Views with SQL Pass Through.”

Record sets created with either remote views or SQL pass through can be manipulated with all the traditional xBase data navigation and manipulation commands and functions, and can be bound directly to controls in VFP forms, just as with native VFP data.

Built-in local data engine

Visual FoxPro has its own built-in local data engine that requires no additional components. Why would you want a local data engine when writing client/server applications? There are three good reasons: local lookup data, metadata and disconnected data.

Some data never or rarely changes. For example, the states in the United States haven't changed since 1959. If data is static and does not require security, there is no particular reason to store it in a server database and no need to send it back and forth across the wire every time a user needs it. So why not keep some of this static, frequently used data on the local workstation? Visual FoxPro's local data engine allows this data to be stored locally, where it can be accessed quickly and frequently with no drain on the network or the server. Just in case this data does change, you can keep a master copy on the server and simply check to see whether it has changed whenever the application starts up. If it has changed, download it from the server and refresh the local copy; otherwise, just use the local copy. This topic is covered in greater detail in Chapter 9, "Some Design Issues for C/S Systems."

Metadata is data that describes other data. Metadata is usually used by the application, rather than by the user. Using metadata in combination with data-driven (rather than code-driven) techniques allows you to create more flexible applications more quickly. If the same or a similar action must be performed on many different items, you can either hard-code the particulars of each item, or you can write a generic routine and then create a table with a record for each item. Adding and deleting items is as simple as adding and deleting records in a table, and reordering items simply requires changing physical or logical record order. Sometimes this metadata should be available to users, but other times it's handy for it to be unavailable.

The VFP local data engine also allows metadata to be joined in queries with client/server data or other local data, even if the metadata is compiled into the EXE. Consider the example of an application that uses metadata to represent rules the federal government has imposed on completion of data entry. Users are also allowed to create their own rules. Since the user's rules mustn't clash with the government's rules, the user is only allowed to apply rules to columns in the database for which there are no existing government rules. The SQL Server database is queried for a list of fields and exclude columns with rules in the metadata table.

A final benefit of VFP's local data engine for client/server development is for disconnected record sets, such as data on laptop computers that are taken on the road and are not always connected to the server. A copy of some or all of the server's data is stored locally. The system can work on this data even while the laptop is disconnected from the server. With Visual FoxPro, you can create disconnected record sets either using the offline view feature or by copying record sets to tables. If local data weren't supported, then another data engine, such as MSDE, would have to be installed and used.

Support for other data-access technologies

While this book concentrates on developing client/server applications using Visual FoxPro's built-in data access technology, VFP also supports the use of other data access components such as ADO. Now you might be wondering why, after all this talk about the advantages of Visual FoxPro's built-in database support, are we talking about using something else? For a traditional two-tier client/server application, there's no particular reason to worry about ADO. But in a three-tier application, ADO has one very strong feature you might want to look at: Data can be passed from one component to another as an object. If a front-end component needs to pass user-modified data to a data-validation component running in Microsoft Transaction Server on another machine somewhere, you have to get the data there somehow. It is very easy to send it in an ADO RecordSet object. The front end could either use VFP's built-in data support and then convert it to ADO, or VFP could just use ADO in the first place,

whichever works best in the specific situation. ADO is covered in more detail in Chapter 12, "ActiveX Data Objects."

Rapid Application Development (RAD)

Rapid Application Development, or *RAD*, means many different things to many people. Visual FoxPro is a great RAD tool for two reasons: Prototypes can be created quickly and turned directly into parts of an application, and VFP is just about the fastest way we know of to build applications.

Prototypes of application components are tremendously useful in the development process. Both developers and users get an opportunity to see what forms will look like and get a feel for the work flow. Working prototypes really help improve a design, particularly when users can work with them. A key word here is *working*. Looking at screen shots just doesn't work as well as working with actual forms, filling in fields and clicking buttons. Many C++ development shops like to use Visual Basic or Bongo for creating prototypes. But then they have to throw away their work and do it over again in C++. Yes, it is still faster than trying to prototype in C++, but wouldn't it be nice if you could simply put the finished prototypes into a project and add code to them? You can do this with Visual FoxPro. Visual FoxPro is as good at prototyping as either VB or Bongo, but unlike VB or Bongo, the finished prototype is fully usable Visual FoxPro code.

We can't quantify Visual FoxPro's development speed, but Gary relates one experience that demonstrates just how fast it can be: "My company has two development teams, one working in VFP and one in Java. When I started on a major project, I was a year behind the three-man Java team. There was just me on the VFP team to develop a similar application. They had more than 400 SQL Server stored procedures written already. Since my application had to work on a VFP back end as well as SQL Server, I couldn't use any of those stored procedures and had to reproduce all the functionality on my own. For one very complicated area, I even attempted to duplicate the object strategy used in the Java version in the hope that it would save me time, as I had quite a bit of Java experience. After working with it for more than a month, I threw it out completely and did my own from scratch. Despite that lost month, *my project was completed nine months before the Java team's!* I would love to say this happened because I'm faster than a speeding bullet, but the truth is that Visual FoxPro made me look like a star. Considering their manpower and time allotment, Visual FoxPro allowed me to complete my application nine times faster and produce six times the revenue at about one-tenth the cost."

Summary

We believe that Visual FoxPro is the finest client/server rapid application development tool available today. And considering that Visual Basic isn't object-oriented (well, not yet—VB 7 promises some level of object-oriented programming), anyone using VFP for client/server development has an automatic advantage over more than half of all developers. Hopefully this chapter has given you some ammunition you might need to support your choice of development tool.

In the next chapter, you'll learn the basics of Microsoft SQL Server.