Pro Visual Studio 2005 Team System Application Development

Steve Shrimpton

Apress®

Pro Visual Studio 2005 Team System Application Development

ISBN-13 (pbk): 978-1-59059-682-1

ISBN-10 (pbk): 1-59059-682-X

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

Lead Editor: Ewan Buckingham
Technical Reviewer: Riley Perry
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Beth Christmas
Copy Edit Manager: Nicole LeClerc
Copy Editor: Liz Welch
Assistant Production Director: Kari Brooks-Copony
Senior Production Editor: Laura Cheu
Compositor and Artist: Kinetic Publishing Services, LLC
Proofreader: Lori Bring
Indexer: John Collin
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `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 Source Code section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 1

Introducing Microsoft Visual Studio 2005 Team System

Software development today seems both easier and more difficult than ever before. It is easier because the tools now available provide developers with the ability to construct complex applications more quickly, yet it is more difficult because defining what those complex applications do, how they are architected, how they interact with other complex applications, how they can be tested, and how they can be consistently deployed and maintained still seems an unrelenting problem.

The Need for Team System

Software development is more than ever a team pursuit, and each of these difficult areas is often the responsibility of different members of the team. Inevitably members see things from their own point of view, and it is rare to find a team member with the experience and perception to see across all the boundaries. Teams can easily become compartmentalized, and in the absence of a concerted approach, the development part of the team is sometimes left to its own devices, because management does not have a clear view of its activities.

One can take the view that only the code really matters in the end, and when push comes to shove, this view often prevails. After all, the customer usually does not see or appreciate the use cases, architectural designs, detailed designs, class structure, test plans, project schedules, or anything else concerned with the process; they usually only experience the finished product and judge it based on its ability to do the job for which they buy it. As long as the code works, the customer is almost always happy. Of course, we all know that this simple statement does not really imply that we can neglect the other traditional parts of the software development life cycle. Indeed, for some customers, such as government defense departments, delivery of documents is as much a contractual requirement as delivery of working code.

But the views of the whole team really are needed. Clear statements of requirements will provide the basis for good design and testing. Thorough and clear design will usually lead to accurate code, leaving fewer bugs to be found by testing and fewer trivial issues to be tracked around the code-test cycle. This provides everyone with more time to concentrate on the difficult issues that justifiably could not have been foreseen. Good, readily understandable architecture and code design also lead to easier modification and better maintainability. Adequate test coverage at the unit, integration, and system levels will provide confidence that the code does

what is required and properly implements each component of the design, allowing true progress through the project to be more visible. Repeatable unit tests and system tests will also enable modifications and refactoring to proceed, where needed, with the confidence that undetected defects will not creep in. Good, enlightened project management, in contact with the pulse of the project, will bring the right forces to bear at the right time, minimize risk, and optimize the chances of a successful outcome.

Nevertheless, most people would agree that good code is the most important end product, and this book is about how the limited view of each team member can be successfully combined using Microsoft Visual Studio 2005 Team System. This is a book that gets deeply into the code development, yet emphasizes that the development occurs in a broader team context.

Perhaps the central importance of good code is why the first and primary software development life-cycle tool to come from Microsoft was Visual Studio, in all its variations through all the versions of Visual Basic and Visual C++ prior to .NET, the watershed of .NET and those versions coming afterward, right up to Visual Studio 2005. All of these evolutions of the product, including the new .NET languages, allow the developer to do a better job of producing, in a more efficient manner, code that implements increasingly ambitious functionality. Visual Studio is a highly complex developer's product and is unfamiliar to many other important members of the software development team. Nevertheless, tools have always been needed to perform the tasks of other team members, such as project management, requirements definition, and testing, leaving the developers to use Visual Studio to its best effect.

In Figure 1-1, the ongoing activities involved in the software development life cycle of a typical Microsoft project are summarized with an indication of the tools often being used prior to adopting Team System. Project managers use Microsoft Project to organize the project's work and Excel to organize project data. Business analysts use Excel and Visio to document and visualize the requirements. Architects use Visio to draw class diagrams, data center diagrams, network diagrams, data models, data flow diagrams, and so forth. Testers use Microsoft Application Center Test, a variety of third-party products, and manual methods to perform tests at different levels. The whole team uses Microsoft Word to generate documents. Such documents, along with the source code, are stored in a version control repository such as Visual SourceSafe or a third-party product with richer features. A variety of third-party products are employed to track progress of the project by maintaining a database of requirements, defects, and other work items. Various freeware and shareware products or homemade programs are used to perform formal controlled builds from the source code repository. Since this book concentrates on Microsoft technology, I have not listed the variety of good third-party products currently used in the software development process. To mention only the ones I have experienced would be unjust to those I have not met. Good though these tools may be, used together in the software development cycle they still do not provide an optimal suite that supports an effective process, for reasons that I will now discuss.

Visual Studio Team System represents a departure from the previous evolution of Visual Studio in that it attempts to bring under one umbrella all of the activities shown in Figure 1-1. By activities, I mean a responsibility area of the project performing a particular function to achieve a successful project outcome. These do not necessarily involve people. Source code versioning, for example, I call an *activity* under this definition since it is an ongoing job that continues for most of the duration of the project, and is implemented by some sort of tool. Project management is another example of an activity, one that involves one or more people and some tools. Software development involves a number of people and tools. Figure 1-1 also attempts to show roughly the interactions among these activities. For example, software

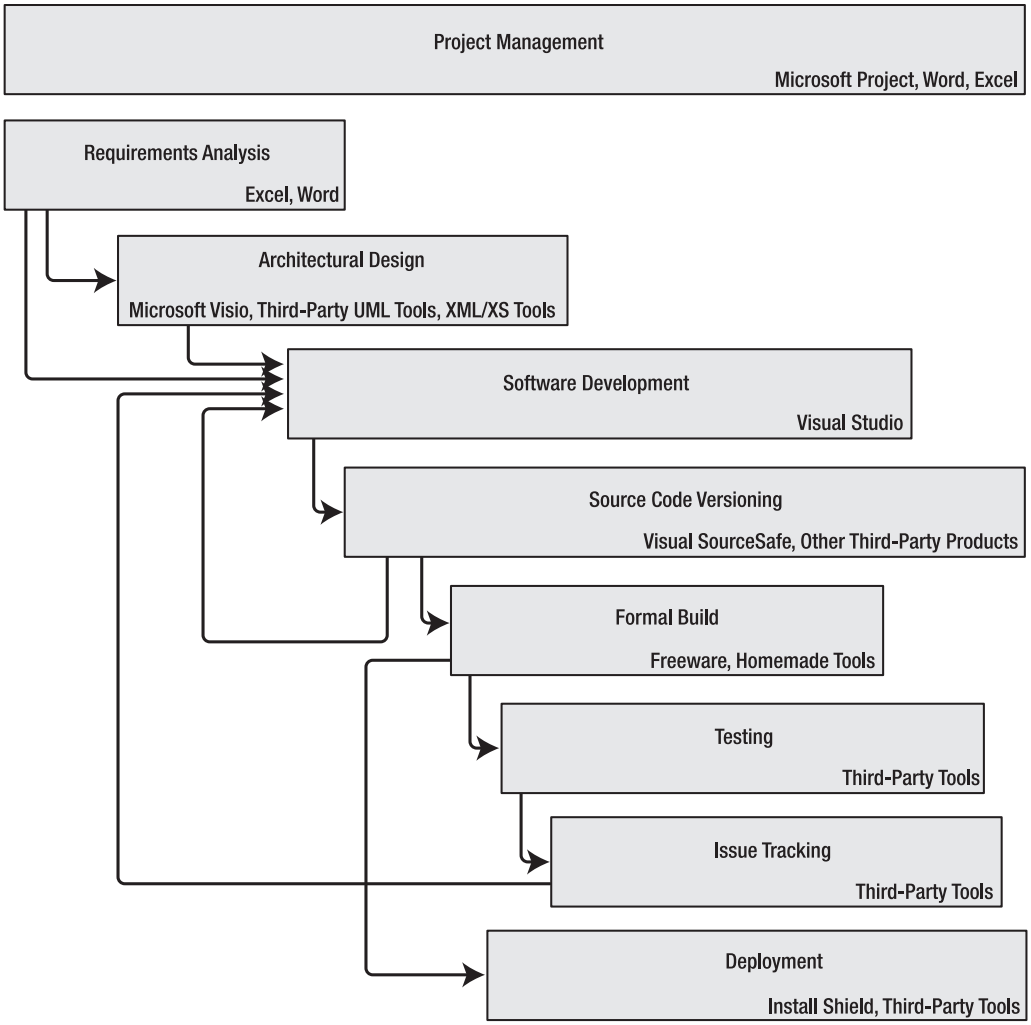

Figure 1-1. *Before Team System: the activities involved in the software development life cycle and their associated tools*

development delivers code to source version control and obtains code from it. Formal build obtains code from version control, and so forth. Not all of the interactions are shown in the figure.

Why would bringing these activities under one umbrella be a desirable thing to achieve? There are a number of reasons. In the current multiple-tool, multiple-vendor environment, all these tools need to be purchased, licensed, installed, and maintained on the team members' machines. Some third-party tools have expensive and complicated licensing requirements. The team members need to learn to use each of these tools and see how they fit into the project and project development process. This can take weeks for new team members, and inadequate knowledge often leads to confusion and inconsistency across the team in the way the tools are

applied. In my experience there is an accumulation of project "folklore" that has to be picked up by new team members when learning how to do such things as performing a formal build, interpreting coding standards, checking in modified or new code, or tracking defects.

The process followed by the project is often another source of confusion and inconsistency. The process is usually described in documents or in a set of process guidance web pages, but is typically described in a general manner rather than in terms of the tools that are actually used to carry out the steps it defines. Thus, it is left up to the team members to interpret the process guidelines and determine the way in which they use the tools to carry out their everyday tasks.

Lack of integration is a third problem with the multiple-tool environment. This affects both the efficiency of the team members and the visibility of the management. Too often in projects, the metrics used by the management to track progress are not really effective, since they are too far removed from the development process and are too difficult to keep up to date. A Microsoft Project report is only as accurate as the data typed into it, which is often communicated verbally by the team members, who will provide such statements as "Feature X is 60 percent complete." Such statements are obviously subjective because seldom does anybody define what 60 percent really means. Does it mean 60 percent of the code is written, 60 percent of the code is working, 60 percent of the scenarios have been demonstrated, or 60 percent of the allocated time and budget have been consumed? Even if we make our statements more precise by saying, for example, that 60 percent of the documented scenarios have been demonstrated, the possibility is still left open that the developer has left the most difficult ones until last! To make such statements meaningful requires not just an integrated tool, but a more precise management discipline among all concerned, for example, to consider each of the scenarios thoroughly, assess their level of difficulty, break them into quantifiable units, and ensure that they are properly demonstrated. This takes a greater level of commitment to management overhead than many project budgets and schedules are able to tolerate, but the cost and time of achieving effective management visibility at the micro level will be lessened by the use of integrated tools. This is because these tools relate code changes, as they are added to version control, to tasks that have been scheduled to implement functionality and fix defects. Up-to-date reports can then be generated quickly from this source of low-level data so that the tracking of task completion against the schedule can be followed more easily.

Consideration of requirements and tasks brings us to one of the key features of Team System: the concept of *work items*. Defects, tasks, requirements, issues, and so forth are recorded in Team System using work items, and they allow the related code to be tracked as it is written and tested. This is the way in which the "Issue Tracking" activity in Figure 1-1 is brought under the Team System umbrella and integrated with the development product. Work items are central to everything in Team System, and I will show their relevance and use throughout this book.

To summarize, in this section I have surveyed the background of Microsoft Visual Studio 2005 Team System and indicated the reasons why a more integrated system of software development life-cycle tools is desirable. It is my task in the chapters that follow to show you how this new integrated system of tools will make your life as a software development team member so much easier. First, however, I must introduce Team System as a product.

Microsoft Visual Studio 2005 Team System

Visual Studio 2005 Team System (VSTS) is aimed at development teams ranging from 5 to 500 members and consists of a set of editions of Visual Studio 2005, each with features intended for use by a particular role in the development team, together with a server application called Team Foundation Server (TFS). The three roles covered are architect, developer, and tester, and there are separate versions for each. Also, a full version is available that has the features of all three roles. Team members who are not involved with the code development are supported by the client application of Team Foundation Server, called Team Explorer, and by the ability of some Microsoft Office applications, such as Project and Excel, to integrate with TFS. Team System therefore integrates the whole team, not just the developers. All editions of Team System provide all of the features of Visual Studio 2005 Professional but the role-based versions have additional features as described in the next three sections.

Visual Studio 2005 Team Edition for Software Architects

Additional features in this version are tools that assist the architect in designing distributed applications and the hardware infrastructure to support them. The tools come in the form of the "Distributed System Designers," which support the design of distributed applications—that is, applications made up of interacting component applications. They are particularly aimed at service-oriented architectures, but can support other kinds of applications using their "generic" features and can be customized to support other specific types of distributed applications more fully. There is an Application Designer, a System Designer, a Logical Data Center Designer, and a Deployment Designer.

The use of these tools is studied in detail in Chapter 5 of this book. Essentially though, the Application Designer allows the architect to design distributed applications in terms of component applications, such as Windows applications, web service applications, websites, generic applications, and database applications. The designer allows you to place endpoints on the application symbols, which can be connected to other applications to show how the distributed application is composed of its interacting component applications.

A distributed application, in this sense, is actually a somewhat general concept for which there can be various deployed configurations, called *Systems,* and the System Designer is the tool that allows these configurations to be designed from an existing application.

The Logical Data Center Designer allows you to design a configuration of logical server and client machines, together with their interconnections. You can then use the Deployment Designer to simulate a *deployment* by selecting one of the systems and deploying it on a logical data center, assigning each component application to a logical server. Various constraints on applications and data centers can be defined that are checked for mutual agreement during deployment.

Visual Studio 2005 Team Edition for Software Developers

This edition is intended for developers and includes a set of tools designed for them, containing a static code analyzer similar to FxCop, unit-testing tools similar to NUnit, as well as code-coverage and code-profiling tools.

For those unfamiliar with FxCop, it is a free tool from Microsoft that checks your code against a set of more than 200 rules that avoid known defects and poor design in areas such as library design, localization, naming conventions, performance, and security. The static code analysis tool in Visual Studio 2005 Team Edition for Developers performs a similar check and is fully extensible; it allows you to easily add new rules. You can enable or disable rules according to whether you regard them as important. When enabled, the static analyzer runs after the build, and you can specify that each rule produce an error or a warning, so you can define rules that require compliance for a successful build. Check-in to the source code repository can then be made conditional upon a successful build and also upon the running of the static code analysis tool.

NUnit is an open source, freely available unit-testing framework, which was developed from a similar framework for Java called JUnit and has had widespread use by developers for several years. The unit-testing capability included in Visual Studio allows you to create a test project simply by right-clicking on a class or member function and selecting Create Tests. The project and unit-testing framework is automatically constructed, and you can immediately write the code to implement the required set of tests. The unit tests are collected together under the Test View window. From here you can select tests for execution and review the results.

Once you've written your unit tests, it is important to evaluate the degree of code coverage of these tests. The code-coverage tool will do this for you by giving a percentage indication of the code that has been covered by a set of unit tests; the covered code is also identified in the text view by a green color (the code not covered is indicated by red). Of course, "covered" simply means executed, and it remains for the developer to determine whether the unit tests adequately test the logic of the code. There is a figure called the *cyclomatic complexity*, which is a measure of the number of linearly independent paths through the code, and although all the code may have been executed by a set of tests, this does not mean that all paths have been exercised. Generally the higher the cyclomatic complexity, the more unit tests you need to properly test it, and you also need to verify the correct handling of a range of data values, to ensure that problems such as overflow are avoided.

The code profiler provides the capability to obtain data about the dynamic behavior of the code by determining the frequency at which each function is executed as well as how long it takes, and thus enables developers to make decisions about performance tuning. For example, if a function within the code is executed very rarely, it makes little sense to spend a great deal of time optimizing it, whereas a function in which the code spends a great deal of its time should be made as efficient as possible. The profiler can operate in two modes: sampling and instrumentation. In the sampling mode, the profiler interrupts execution at regular intervals, determines which function the code was executing when interrupted, and maintains a count of the number of times each function was detected. Statistical data is maintained, which over a period of time gives a reasonable profile of where the code spends most of its time. When the instrumentation method is used, the profiler inserts enter and exit probes in each function, which allows it to detect not only each and every time that function is entered, but also how long it takes to execute on each occasion. Although this method is more intrusive and makes the code execute much more slowly, it gives a very accurate profile of the dynamic behavior of the code.

A class designer is also included that provides graphical UML-style class diagrams, which are maintained in a synchronized state with the code. You can modify either the class diagram or the code and they remain in agreement. This feature is actually included in the other versions of Visual Studio 2005 and you do not have to buy a Team Edition to obtain it; however, this

feature forms such an important part of the development described in this book that I will discuss it frequently. The unit-testing and code-coverage tools are also included in the edition for testers since it is not always clear where the responsibility lies for this kind of testing.

Visual Studio 2005 Team Edition for Software Testers

This edition is intended for software testers and includes tools for developing various kinds of tests and for managing them, running them, and reporting the results.

The unit-testing, code-coverage tools, and Test View are the same as those included in the edition for developers, but in the edition for testers, there is an extra feature called Test Manager that allows tests to be categorized into multilevel lists. The version for software testers provides the capability to define test projects that are placed under version control on Team Foundation Server in the same way as any other piece of code. The types of tests that can be included in a test project are unit tests (as in the developer version), manual tests, generic tests, web tests, load tests, and ordered lists of tests.

Manual tests are implemented either as plain-text or Microsoft Word documents, and Word can be automatically launched when a new manual test is created with a standard test format. There is a Manual Test Runner, launched from Test Manager, that enables the tester to record the result of the test. The failure of the test is associated with a new work item. Thus, manual tests are included in the overall testing structure and are integrated into the Team System environment.

Generic tests can be used to wrap around existing third-party automatic tests or test harnesses written as part of the project. To be able to be integrated into Test Manager as a generic test, such a test must conform to a certain data or return code interface.

Web tests allow for the functionality of web pages to be tested by recording a sequence of navigation through the pages. Web tests allow you to validate the results returned in web pages and are thus a functional rather than a load test.

Load tests allow you to simulate an environment where multiple users are interacting with an application simultaneously. A wizard lets you define your included tests and load profile, which specifies how the load varies with time. You can then run the load test and obtain selected information about performance.

An ordered test is simply a list of tests, excluding load tests, that must be run in a specified order.

I will describe the use of these various tests in detail as the book progresses through the various development and testing stages of the project.

Visual Studio 2005 Team Suite

This edition includes all of the features of the editions for architects, developers, and testers in one package, which is useful for team members or consultants who play more than one role.

Visual Studio 2005 Team Foundation Server

Team Foundation Server is the server system that ties the architects, developers, testers, and other team members together into a team. Visual Studio 2005 Team Editions communicate with TFS as a centralized repository for source control, work item tracking, builds, and other project artifacts such as Visio or Word documents. It consists of two tiers:

- A data tier consisting of a set of databases hosted within SQL Server 2005
- An application tier providing a set of web services that are used by the various instances of Visual Studio 2005 Team Edition installed on the workstations of the team members, and by instances of Team Explorer (which we see in a moment), as well as Excel and Project

The application and data tiers of TFS may be installed either on one machine or on separate machines. TFS also contains a project web portal implemented by Windows SharePoint Services (WSS). This portal acts as a place where team members can disseminate information to each other in the form of announcements; list and view useful websites; obtain reports of the project's status; and view development process guidelines. It also provides a repository for documents under control of WSS to provide versioning, check-in, check-out, and integration with Microsoft Office applications such as Word, Excel, PowerPoint, and Project.

Included with TFS is Team Explorer, which is a lightweight client for TFS that enables team members not directly involved in architecture, development, or testing to access documents on the server and create and manage work items. As an alternative to this, such team members can also use Excel and Project to access and manage work items. We will study all of these tools in detail later in this book.

How Team System Is Described in This Book

This book will do more than describe the features of Team System; it will show how it can be used in a real project that involves developing a significant distributed, service-oriented application that forms the core enterprise system of a sizable business. This is exactly the kind of project for which Visual Studio 2005 Team System is intended, and I will trace the project through its various stages and show how the features of Team System are used. For example, I will show how the project management tools are used to plan and track the progress of work items. I will also show how the architecture tools—the distributed system designers—are used during the architectural design phase of the project. As the project moves through the build phase, you will see how the code development, code analysis, and test features can be utilized to achieve stability.

The project, of course, is imaginary, as is the company and characters, but I will use their business and staff members to illustrate the interactions among team members, project tasks, and artifacts as the development moves along. The project is fairly straightforward and is typical of many kinds of applications that are developed across the world. I think a completely innovative project, using new techniques, would be too complex to describe in the depth necessary, and you would doubtless become too involved in understanding the complexities of the code to appreciate the bigger picture. Nevertheless, I have included some challenging areas that I hope will illustrate the iterative and risk-reducing way in which I have shown the management of the project.

This brings me to my second aim, which is to show how Microsoft Solution Framework (MSF) can work in a real project. MSF is Microsoft's framework for the software development process. Currently two variants ship with Team System: MSF Agile and MSF for CMMI Process Improvement, which I discuss in Chapter 3. I have chosen MSF for CMMI Process Improvement since this more formal approach, based on the Software Engineering Institute's Capability Maturity Model Integration (CMMI), is more appropriate to a project of the type illustrated, where the company's business is very dependent on the success of the project and risk must be kept to a minimum. Of course, I have been unable to show every detail and document produced by the

team as they follow the process—such an approach would bore my readers and be beyond the capability of my one pair of hands—so I have tried to give a representative picture of how the process is applied by examining selected artifacts at various critical points in the development.

Similarly, I have not been able to generate all of the requirements, design, and code for the project but have included a representative sample. I do not expect you will want to check every line and verify that the project is complete! However, I have shown the complete implementation and testing of certain critical areas of functionality, which you should be able to build, test, and run at the end of each stage of the project. Indeed, a third aim of this book is to illustrate an architecture that can be used for a typical distributed service-oriented application. I have tried where possible to use architectural techniques that are recommended by Microsoft, but I have also been inventive in certain areas—which, I claim, is how a good architect should work—and of course I show only the architectural technique that I have chosen with limited time available, which works in the scenario illustrated. I have attempted to choose architectural techniques that illustrate the new features of ASP 2.0 and Visual Studio 2005, and also show some diversity; for example, I have included both browser-based and smart clients, for convincing architectural and business reasons, I think. You may well be able to think of better techniques and indeed in a real project, you should invest time in evaluating alternative architectures to find the best. However, remember to invest such time in areas that matter, where the architecture choice impacts a key feature of the system such as performance or security. In many places in the code, a suboptimal design may be quite acceptable as long as it is clear and works in a satisfactory manner.

I have used the terms distributed application, service-oriented architecture, and smart client in this section, and these concepts will permeate this book. I daresay you are familiar with them to some extent, but I will spend the remainder of this introductory chapter reviewing their meaning and placing them in the right context for the ongoing discussions.

Distributed Applications

The idea of a distributed application is not new, and has appeared in many forms ever since computers became cheap enough for the move away from the centralized mainframe model to take place. I'm going to define it this way: a distributed application is a collection of software components, working together to implement some defined business capability; each of these software components has some degree of autonomy in that each is either physically or logically separate. The whole distributed application may use a number of applications that can be considered complete in themselves, which we can call component applications. I'm sorry to use the word "component" here because that word has a special meaning under COM and DCOM that I will mention later. I do not imply that meaning here, and I could just as well use the term "building block" as I simply mean applications that work together to make bigger applications.

To constitute a distributed application, the software should be under some degree of management as a whole. A set of component applications that just happen to work together one day but fail later because one of them has been changed could not be considered a distributed application. An example of this is an application that uses a number of uncontrolled websites to obtain trading information about securities by screen scraping. *Screen scraping* is a technique used to extract information from the HTML presented by a website intended for viewing by users. If the distributed application no longer works because of an arbitrary change to the screen appearance by one of the websites, then the application cannot be considered a distributed application under this definition.

The point, then, about a distributed application is that the component applications must be controlled in some manner so that they can always be relied on to work properly together. This does not mean that they must be developed by the same team, or managed by the same managers, or even owned by the same company. All that is required for a distributed application to operate properly is that the interfaces exposed by the component applications must be agreed on and maintained. What does the phrase "exposing an interface" mean? It is rather like offering goods for sale. The component application advertises a contract that may be entered into by a purchaser. The purchaser does something and the vendor does something in return; the contract is agreed on and both sides know what they are getting into. If I go into a grocery store and buy onions, I expect to have onions in my basket—I do not expect them suddenly to be replaced by mangoes; similarly, a software contract must be clearly defined and unchanging.

Those who are familiar with service-oriented architectures may see that we are moving along toward defining providers, consumers, WSDL, SOAP, and so on, but let's not jump ahead. Distributed applications are not necessarily service-oriented architectures; there are other forms, and I'm going to mention some of them in order to give a wider illustration of the software world.

Let's start with something completely different. Military aircraft in the Western world contain various computers that are manufactured by different vendors and perform part of the job of flying the plane and doing battle. Examples are Mission Computer, Fire Control Computer, Stores Management System, Inertial Navigation System, and so on. I won't describe what these do, other than to say that *stores* are things like bombs and rockets, but you can get an idea from the names. In the 1980s, these computers used to communicate with each other using something called a serial data bus, which transferred data between one computer and another according to a predefined schedule. For example, every so often a message would be initiated that would transfer data from the Inertial Navigation System to the Mission Computer, which would update information about the location of the airplane and be used to update navigation displays. The system is shown in Figure 1-2.

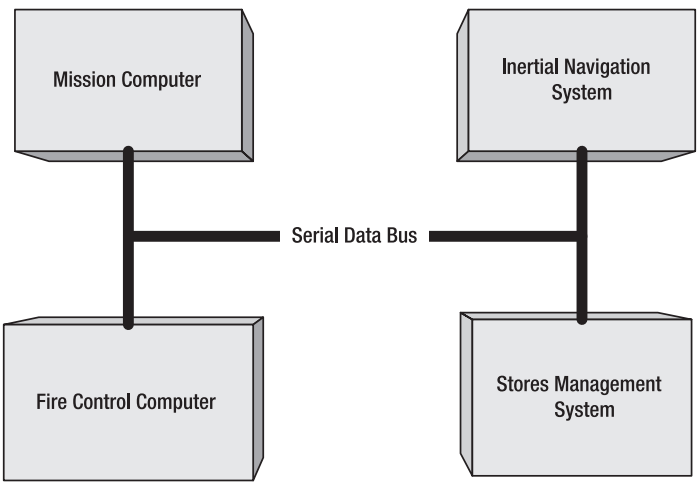

Figure 1-2. *A distributed application on a military airplane*

Is this a distributed application according to our earlier definition? Yes, it certainly is. The whole set of component applications in the various computers combine to form an application whose job is to allow the crew to perform their mission. The component applications are not under the control of one management team or one company, and each can be modified independently. The only criterion is that they support the contract between them. Think of the Inertial Navigation System; it must always respond to a predefined command in the same manner, giving the location of the airplane in an agreed format that the other component applications can interpret correctly. There is another side to the contract, though, which we can call nonfunctional requirements, such as the requirement that the application in the Inertial Navigation System always responds within an agreed time.

Can you use Microsoft Visual Studio 2005 Team System for this distributed application? Well, theoretically I think you could, but you would have to customize the Application Designer and you probably would not be using managed code such as C# or Visual Basic for these embedded systems. So it's a bit of an extreme example!

Let's move on to something closer to home and consider a different kind of distributed application. Before .NET was invented, people used to write applications using Component Object Model (COM) and DCOM. COM is a scheme that allows components of an application, perhaps written in different languages, to work together by exposing interfaces containing member functions. DCOM is a variant that allows the components to reside on different machines and expose their interfaces across a network. The code for the components may be contained either within dynamic link libraries (DLLs) or within separate executables. Either way, they can be constructed and maintained independently and can come from different vendors. The component applications can work together, provided contracts are maintained between them. These contracts consist of the defined interfaces that each component exposes for use by other components making up the distributed application. The components are really objects and the interfaces define methods. This is called *component-based distributed computing*, and a distributed application built in this way is a *component-based distributed application*.

Let's imagine an example. Suppose I'm writing a Microsoft Word document and I want to include within it some UML class diagrams. I also want a few mathematical equations and a couple of spreadsheets. First, I open Word 2000 and create a new document and type some text. Then I decide to draw a Visio UML class diagram. I can do this by selecting Insert ➤ Object from the main menu and choosing Microsoft Visio Drawing from the list of objects available, as shown in Figure 1-3. I then get a dialog box, as shown in Figure 1-4, and if I choose a UML diagram and click OK, I see the screen shown in Figure 1-5. I can now add classes to the UML diagram embedded in the Word document just as if I had opened the Visio application independently. If I click somewhere else in the document, the embedded Visio drawing will close but will still show the passive image of the drawing. I can edit the drawing again by double-clicking on it.

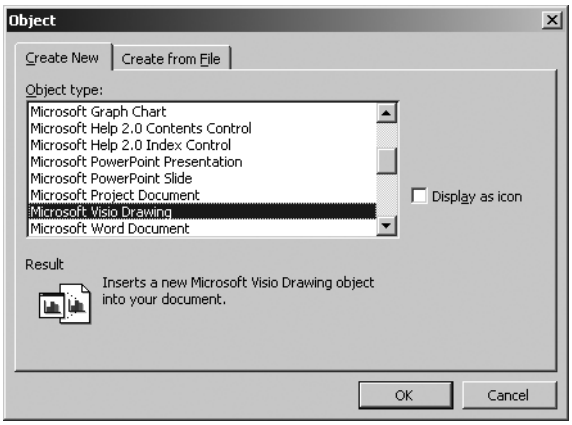

Figure 1-3. *Inserting a Visio drawing into a Word document: choosing the object type*

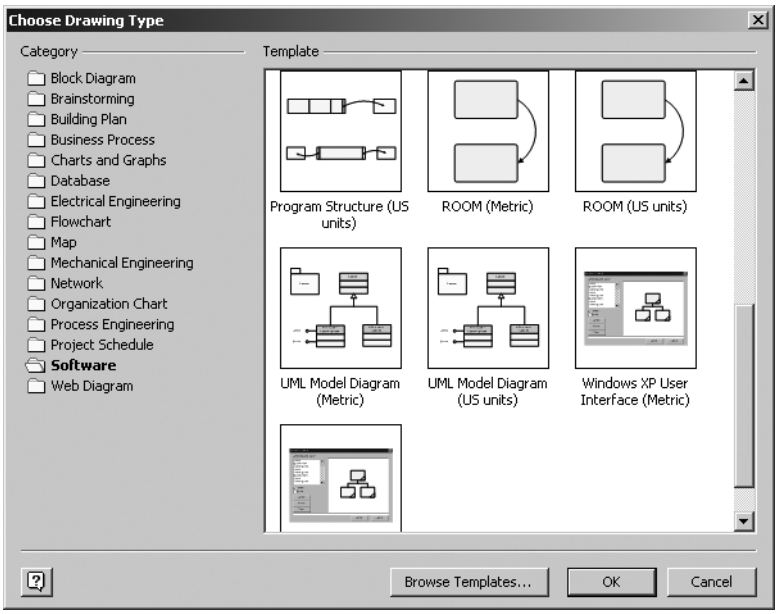

Figure 1-4. *Inserting a Visio drawing into a Word document: choosing the drawing type*

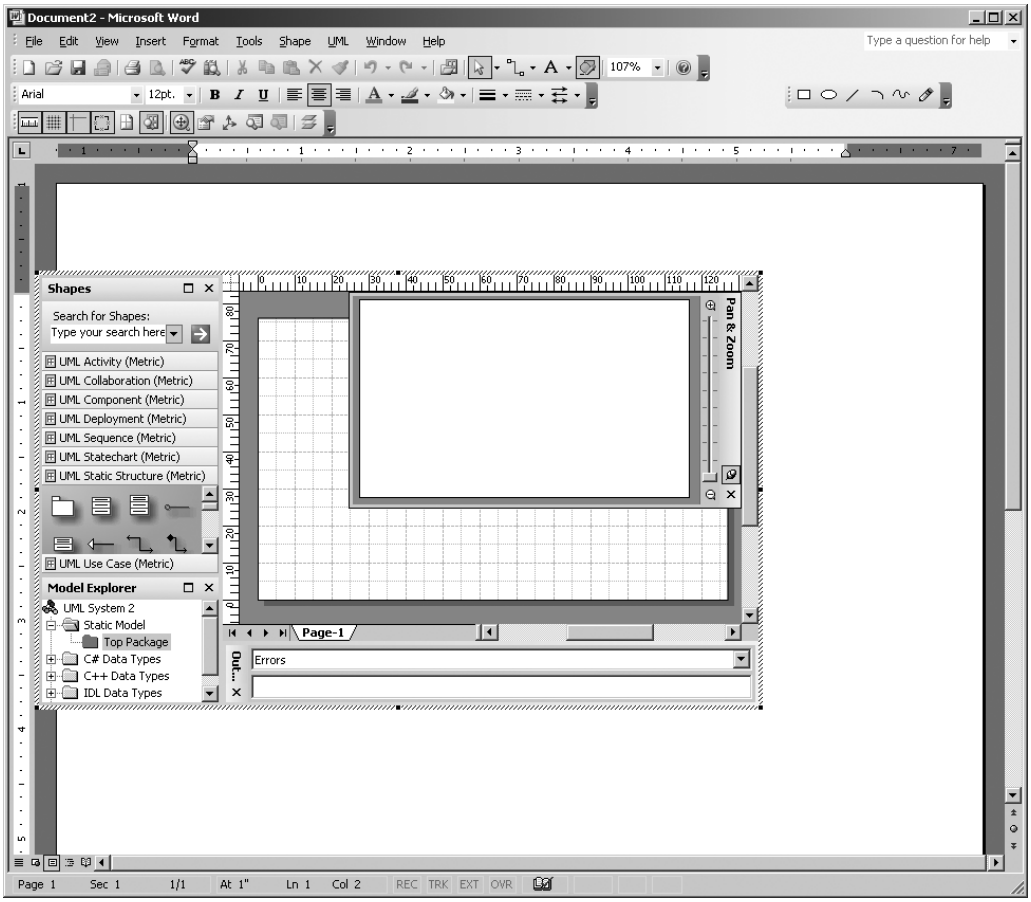

Figure 1-5. *A Visio UML diagram embedded in a Word document*

How does this work? It is all done by a technology called Object Linking and Embedding (OLE), which is a variation of the COM technology that we have been discussing (though OLE 1.0 actually predates COM). It defines the capability of applications—in this case executables called COM servers—to link or embed within other objects, instances of the objects they can manufacture. In the example, the Visio document object is embedded in the Word document object. If you look at Task Manager while the embedded diagram is being edited, you will see that both `WINWORD.EXE` and `VISIO.EXE` are running. Winword communicates with Visio and delegates to it all the work of editing the Visio diagram within the Word document. It does this by calling member functions on a whole host of interfaces that are defined as part of the OLE standard and are implemented by the embedded object. A similar thing happens if we go through the same procedure but insert a mathematical equation object, by selecting Microsoft Equation 3.0 in the Insert Object dialog box. In this case, the application `EQNEDT32.EXE` acts as a COM server, and will start up. A similar process happens when a Microsoft Excel spreadsheet is inserted.

I do not want to get into the details of OLE and COM, although these are still in use and can be included in a .NET application using COM Interop, except to say that these four component applications working together form an example of a component-based distributed

application. The function of the distributed application is to provide the capability to write documents with diagrams, equations, and spreadsheets. The component applications, Word, Visio, Excel, and Equation 3.0, are completely separate applications, but they all expose interfaces that comply with the COM and OLE standards. In this case all of the component applications come from the same vendor, but that is not essential since the OLE standard is well enough defined to allow vendors to write applications, knowing that they will be able to provide objects that can be embedded into other documents. Notice that these applications run on the same machine, so they are not geographically distributed. However, as mentioned earlier, DCOM does allow for that geographically separated case as well.

Can we use Team System to help us write these kinds of distributed applications? The answer is yes, but we would have to customize the Application Designer or use generic applications. Generic applications are catchall applications in addition to those defined in the Application Designer and cannot be implemented—that is, their code cannot be generated, or reverse-engineered—by the Application Designer. We will see how generic applications are used in Chapter 5.

By now you should have a good idea of what I mean by a distributed application, so let's now move on to discuss a particular variety that is built around the concept known as a service-oriented architecture.

Service-Oriented Architectures

Service-oriented architectures (SOA) are component-oriented architectures modified to suit the Internet. An SOA is a distributed application that separates its functional areas into smaller applications that provide or use services. An application that makes available a service to be used by other applications is called a provider, and an application that uses a service made available by another application is called a consumer. To use a service is to consume it. There are two defining features of the relationship between a service client and a service provider:

- Services are loosely coupled to their consumers so that the consumer makes a request of the service, the service responds, and the connection is closed.
- Service providers are stateless, which means that each request to the service is treated as a completely independent operation, and the service has no memory of any sequence of requests.

Service-oriented distributed applications have many of the characteristics of component-based distributed applications. For example, the service applications that are integrated to form the one distributed application may be developed on different platforms by different teams under different schedules, as long as the interfaces by which the services are accessed are agreed on. The difference is in the degree of coupling. A provider and a consumer in a component-based architecture, as in COM and OLE, have a much tighter coupling, which is connection oriented. The provider maintains state, the consumer can make multiple calls on the provider during the connection, and the connection can be maintained for as long as is required. The provider can even make return calls on the consumer, in the form of events, for example. This is the case in the OLE example in the previous section and results in a great deal of complexity.

Also COM, OLE, and the newer .NET Remoting all involve the provision of *objects* to the consumer, which the consumer can hold between function calls and which retain state. In the service-oriented approach objects may certainly exist on the client and server, but these are

simply encapsulations of the message transmission and receipt functionality and do not survive on the provider beyond the transmission of the response message to the consumer. Clearly the movement to a looser coupling is a natural response to the demands of the Internet, where there may be a low bandwidth and unreliable connection between provider and consumer.

The stateless nature of a web service does not, of course, imply that any business entities that are processed by the service have no state. For example, a service may provide a method to modify the details of a business entity, such as a customer held on a database to which the service provider has access. For instance, a request is received by the service to change the customer's telephone number, the request is processed, the data is changed in the database, and a response message is sent. The state of the customer as an entity in the database has certainly changed, but the service itself has no memory of the transaction—there cannot be a sequence of messages that together cause the customer data in the database to change. This is what we mean by stateless. Furthermore, we can regard the customer existing in the database as an object, but that object is not exposed to the consumer as such, other than perhaps implicitly via the message interactions defined in the web service.

These statements define a service-oriented architecture, but there are further characteristics applicable to .NET web services. The first is that the services are accessed via the sending and receiving of messages. A service provider exposes a service interface to which all messages are sent, the consumer sends a message to the provider requesting the service, and the provider responds with a message containing the results of the request. The second is that the "web" in the name web service implies that HTTP will be used since this is the ubiquitous protocol of the Internet and is invariably passed by firewalls. The third characteristic applying to most .NET web services is that the data included in the request and response messages is formatted according to the Simple Object Access Protocol (SOAP), which is defined in terms of XML structures.

Although the concept can be interpreted more generally, in this book I will treat an SOA as a collection of applications that are loosely coupled as either providers or consumers (or both) of web services and distribute the functions of the application among those providers and consumers in such a way that the data relating to a particular service is encapsulated locally with that service. In many applications, though, it is not true to say that any given service exclusively uses the data tables in the database that it accesses. This might be an ideal, but in practice the database often forms the central part of a large distributed application and many services access the same tables for various reasons. Care must then be taken to set up a mechanism of transactions to ensure that the integrity of the database is maintained.

A further feature of web services that needs to be mentioned is that their details can be *discovered* by a consumer. Each service can give to a consumer details of the service it provides in the form of a Web Service Definition Language (WSDL) file. A potential consumer can request this file, which provides complete details of the services offered, as well as the request and response message structures, including definitions of any parameters, and their complex types if applicable.

Figure 1-6 shows an example of an SOA implementing a sales website. The client is browser based, running on a customer machine. The browser client accesses the Sales Website Application, which accesses a database to keep a record of customer orders, and also uses the functionality provided by the Catalog Web Service, giving a listing of the company's products and their prices. The Catalog Web Service also uses a database that contains tables listing the products offered for sale, and uses another web service, the Currency Conversion Web Service, which provides information to convert prices into the currency requested by the customer. This web service makes use of a third database that contains currency conversion factors, which are updated daily.

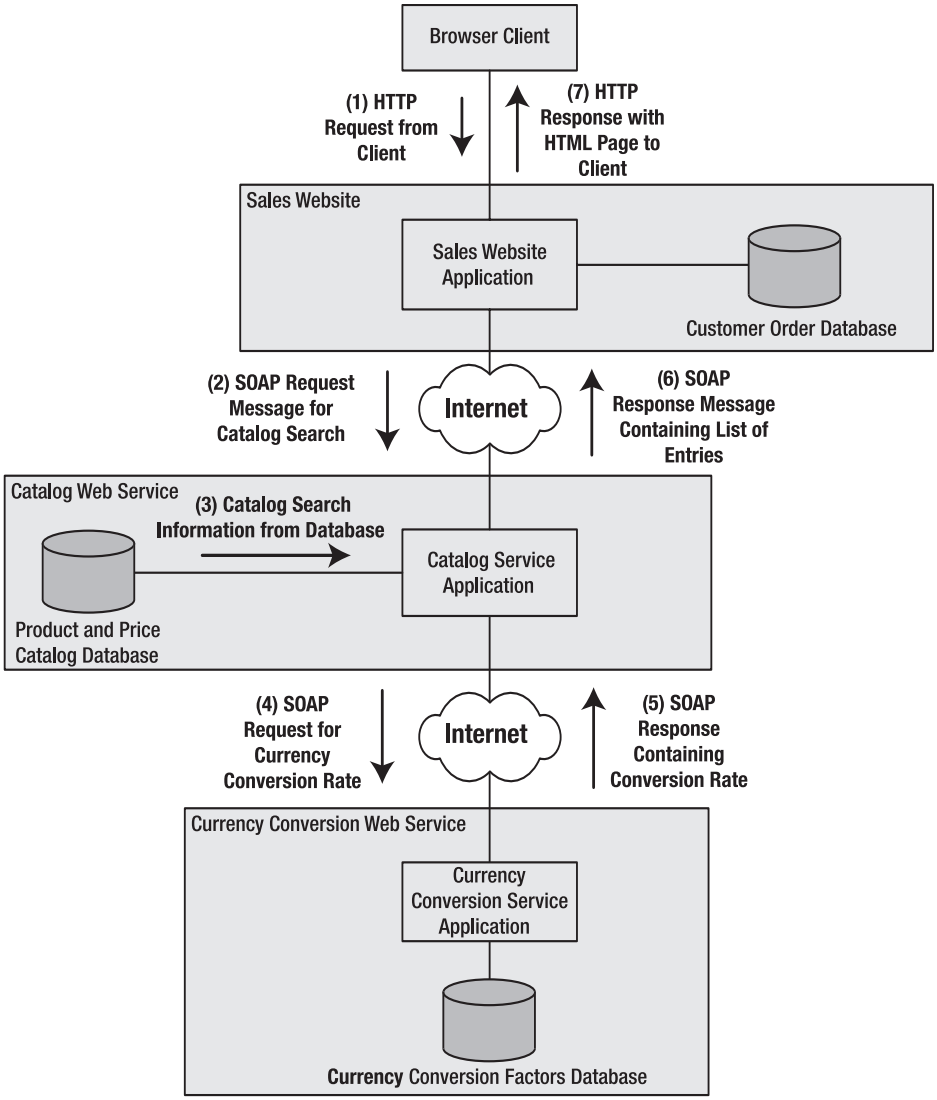

Figure 1-6. *A typical service-oriented architecture distributed application: a sales site using a catalog web service and a currency conversion web service*

Figure 1-6 shows the message flow for a typical search of the catalog initiated from the browser client:

1. The browser client posts a page to the Sales Website Application to initiate a catalog search.

2. The Sales Website Application processes the post and determines that a call to the Catalog Web Service is required.

3. The Catalog Web Service receives a request message for a catalog search and calls a stored procedure in the Product and Price Catalog Database to execute the search. The stored procedure returns a list of catalog entries, each with a price in U.S. dollars.
4. The Catalog Service Application makes a call to the Currency Conversion Web Service to request a conversion rate between U.S. dollars and the user's currency preference, which happens to be Indian rupees.
5. The Currency Conversion Web Service returns a message containing the required currency conversion factor.
6. The Catalog Service Application converts all of the prices in the list of items from the catalog to rupees and returns the list of items in a response message back to the Sales Website Application.
7. The Sales Website Application populates a Web Forms page with the data and returns an HTML page to the client browser.

The Smart Client Approach

The architecture of enterprise applications has evolved over time. Client-server architectures have given way to browser-based, thin-client designs, and now the *smart client* is gaining vogue. Let's now examine this evolutionary story and explore the reasons behind it.

Evolution of the Client-Server Architecture

During maybe the last 10 to 15 years, the architecture of enterprise applications has moved through various stages. We won't discuss small-scale single-user applications running on a single machine, where all of the data and software is contained within that one machine. Here we'll focus on the Microsoft array of technologies; I want to trace the reasoning behind their evolution, although actually the evolution of these technologies seems to be fairly representative of the evolution of the industry as a whole, since there has been such a degree of exchange of ideas between say, the Java and .NET schools of thought.

Once multiuser enterprise applications moved away from centralized mainframe-based applications using terminals such as 3270, and Microsoft solutions began to be adopted as well as others, two common categories of architecture emerged. Both of these can be considered client-server in a manner of speaking.

One type of application uses a file server as the server side and places all the application's software on each client machine. An example is a Microsoft Access application where the database files reside on a server and are opened by the clients. Another is an application in which flat files are stored on the server and each instance of the application opens the file to read, or opens and locks it to write. These are called "thick-client" applications, where all of the business logic is duplicated on each client machine.

The other type of client-server application uses a relational database on the server with stored procedures and triggers. The stored procedures can be very simple, or they can be complex and include business logic. An example of a stored procedure that includes business logic might be called `AddSalesOrder`. `AddSalesOrder` can receive raw textual information taken straight from that entered by the user on the client screen. This stored procedure checks the data for validity, and then performs multiple inserts into various tables to add a sales order, perhaps

obtaining more data by making joins across other tables or even attaching to other databases. A simple stored procedure might just add a row to the SALESORDER table and be given the exact values for each column. You could design this kind of client-server application either with most of the business logic on the client or most of it on the server. Both approaches have advantages and disadvantages. Putting the business logic on the server means that the number of times the client accesses the database is usually minimized, and it also means that the client can be made much simpler. It probably reduces network traffic at the expense of making the database server do much more programmatic work in running the stored procedures, but if the database server is a big, powerful machine, this may not be a problem.

Most people would agree, however, that while a good SQL programmer can make the language do almost anything, SQL is not intended to perform logic but to access data. So if there is a lot of decision making to be performed in the business logic, it is better written in the language of the client, such as Visual Basic or Visual C++. If you are prepared to accept business logic written in SQL, though, this kind of architecture can separate the business logic from the display logic, and thus represents a two-tier system.

The client is still what we consider to be "thick" in the sense that, in the Microsoft world, it is a Visual Basic or Visual C++ application running as an executable on the client and has a set of DLLs that will be required for it to operate properly. Here is the problem! The well-known phrase "DLL Hell" describes the situation in which several applications installed on a client expect different versions of a given DLL. You could install a client application that uses various DLLs that come with the operating system and everything works fine, but then later someone installs another application that overwrites those DLLs with different versions—maybe earlier ones—and suddenly the first application breaks. The first application is reinstalled, but it still does not work because it relied on DLLs that were installed with the operating system. The only answer is to reinstall the operating system—what a nightmare!

Another scenario is that the developer builds an application, installs it on his own machine, and it works fine, but when someone else installs it on a different machine, it does not work. This is because the developer happened to have DLLs installed on his machine as a result of installing Visual Studio. So now the team has to try to find which DLLs are on the developer's machine but missing from the client machine. This situation is made worse by the fact that DLLs load other DLLs in a chain so that even though an application may have access to the correct version of a required DLL, that DLL may require a second DLL that also must be the correct version. The task of solving this problem then becomes more complicated as a chain of DLL versioning requirements must be satisfied.

In all these examples, careful control of client machines is needed to ensure that the suite of applications installed is compatible. When there are several client machines in a company, this becomes a serious expenditure of time and labor. Even leaving aside the "DLL Hell" problem, installation of a complex application on client machines together with possible hardware implications is still a significant task. The same rationale applies to any Windows application built using technology from other vendors such as Borland's Delphi or Sybase's PowerBuilder, since these use many DLLs installed with the Windows system and are subject to versioning. Thus, there is a justification to move away from the idea of having large Windows applications on client machines, and thin-client applications have become a popular alternative.

Thin Clients and the N-Tier Architecture

The World Wide Web began with websites supplying textual and pictorial data, which was displayed by users on their client machine using an application called a web browser. Thin

clients developed as websites grew from simply supplying data for display, to permitting customer input via HTML forms, with controls such as drop-downs, text input, and buttons rendered by an enhanced browser. A telephone directory site is a good example of a website that allows simple customer data entry in the form of search criteria. Gradually the concept of e-commerce came into being, with customers placing orders for goods and services via the Internet using their browsers. The web browser, when displaying HTML forms that form part of the graphical user interface to an application, is called a thin client. One of its chief features is that the architect of the server part of the application neither needs nor has much control over the client configuration.

It became clear to architects of enterprise applications that this technique could be used within the bounds of the corporate intranet. Employees responsible for processing company data could be connected to a web server using only their browsers. A range of corporate applications could then be hosted on that server and be operated by employees with a very simple and standard client machine configuration.

Thin clients then, are characterized by the use of some kind of general-purpose display application, invariably a web browser, and receive all the formatting and display detail from the server. The web browser accesses a sequence of URLs on one or more servers to obtain the screens that the user will see when using the application to perform its business functions. Since Microsoft Internet Explorer is such a ubiquitous application, and is installed as part of the Windows operating system, we can usually assume it is available. Provided the application that sends the HTML to the client for each screen does not use any features that are specific to a particular version of Internet Explorer, the application will always work. This technique has the advantage that a home user—over whose computer the architect has no control—can also run the application.

The server that delivers the business application to the client can run an ASP, or more recently an ASP.NET application, and becomes a *middle tier*, with the browser client forming the top, or client tier, and the database forming the bottom, or data tier. In a simple ASP application, there are two possible places to write the code that implements the business logic. One is to program it into stored procedures in the database in a similar manner to the client-server application described previously. The other is to implement it using JavaScript or VBScript embedded in the HTML pages stored on the server, and enclosed within special tags. The web server includes a script engine that executes this code on the server before delivering the HTML to the client browser.

For more complex ASP applications, the middle tier is broken into more than one tier (the n-tier model) and the business logic is encapsulated in COM components in the form of objects delivered by COM servers in response to create requests from the script in the ASP pages. Various member functions of these objects can then be called by the ASP script to execute business logic and database operations that are coded in the language of the COM components—either VB or C++. The set of ASP pages then becomes known as the presentation tier and the COM components are called the business tier. In a good design, the ASP pages consist of only simple script and the HTML that produces the required appearance, while all of the business functionality of the application is placed in the COM components or the stored procedures in the database.

With the advent of .NET, COM components have become unnecessary for this type of application, and the business logic can be separated from the presentation code quite well by placing it in the code-behind classes in the form of C# or VB.NET code. For a simple application, this is very good, but suffers from the disadvantage that areas of business logic needing to

process data relating to more than one web page must be duplicated in the `aspx.cs` files, and thus become more difficult to maintain and test. It is better therefore to reintroduce a form of n-tier by placing business logic in appropriate classes that can be unit-tested independently, and that can be used by many code-behind classes corresponding to many web pages. It is also a good idea to place the data access code either in these business object classes or, as a much better option, in a separate tier again called the data access tier or data access layer (DAL). This separation of presentation, business, and data access functionality is usually seen as good practice.

OK, so why do we need anything else? There are a few problems with thin clients; for one thing, it is seldom possible to avoid putting some sort of logic on the client in the form of client-side scripting executed by the web browser. This scripting on the client, of course, is not installed on the client like a thick-client application, but is delivered by the web server with each page, and executes for various reasons on the client browser before the page is posted back to the server. One example is when the options in one drop-down control depend on the setting of another—like a list of states in an address depending on the selection of a country—so that when one is changed, the collection of options in the other needs to be updated. This script is never easy to debug and maintain, and its success often depends on a particular version of the browser.

As well as these problems with scripting, the display features that can be presented on a browser client are rather limited. To provide a richer client experience, the designer is invariably drawn back to using a Windows application as the client tier.

Smart Client Applications

A "smart client" application is the name that Microsoft uses to define a client that is a compromise between a thick client and a thin client. It is a full Windows Forms application that makes use of both local resources and network resources, and can operate as part of a service-oriented architecture. It is distinguished from the old-fashioned thick client in that it usually does not interface directly with a database but with a server that exposes one or more web services or other network resources. The smart client application is intended to take the presentation tier away from the server and combines it with the client tier, formerly implemented with a browser, to provide a richer client experience.

Figure 1-7 shows how a smart client is used in a service-oriented architecture. The smart client is a Windows Forms application installed and running on the client machine. Here's how it might work. It might look like a multiple-document application (an application in which several documents can be open on the screen at one time), although the "documents" will probably not be documents stored on the local drive but represent entities stored in the remote database. Let's suppose it's a customer relationship management system. It might have a File menu that has an Open Customer option that allows the user to view the details of a customer by choosing that customer from a list. Imagine the user clicks the Open Customer menu item, which brings up a dialog box listing the available customers. To populate the list in the dialog box, the application calls a function on one of the web services exposed by the ASP.NET application in Figure 1-7. There is a web service, perhaps called Manage Customers, which will have a set of functions defined. One of these functions, `ListCustomers`, returns a list of customer names that the smart client application can display in a list box on the Open Customer dialog box.

When the user has selected the required customer from the list and clicked OK—or maybe just double-clicked the name in the list—the dialog box disappears and the customer details are obtained from the server to be displayed in the application as an opened customer. To do this, the smart client needs to make another call on the Manage Customers web service, this

time perhaps calling a function called `GetCustomerDetails`. In each call to a web service function, an XML SOAP document is sent to the server and another SOAP response document is returned containing the required information. Code within the smart client is responsible for populating the various controls on the screen, such as the list box displaying the customers and the various labels, text boxes, and combo boxes displaying the data fields representing the customer details.

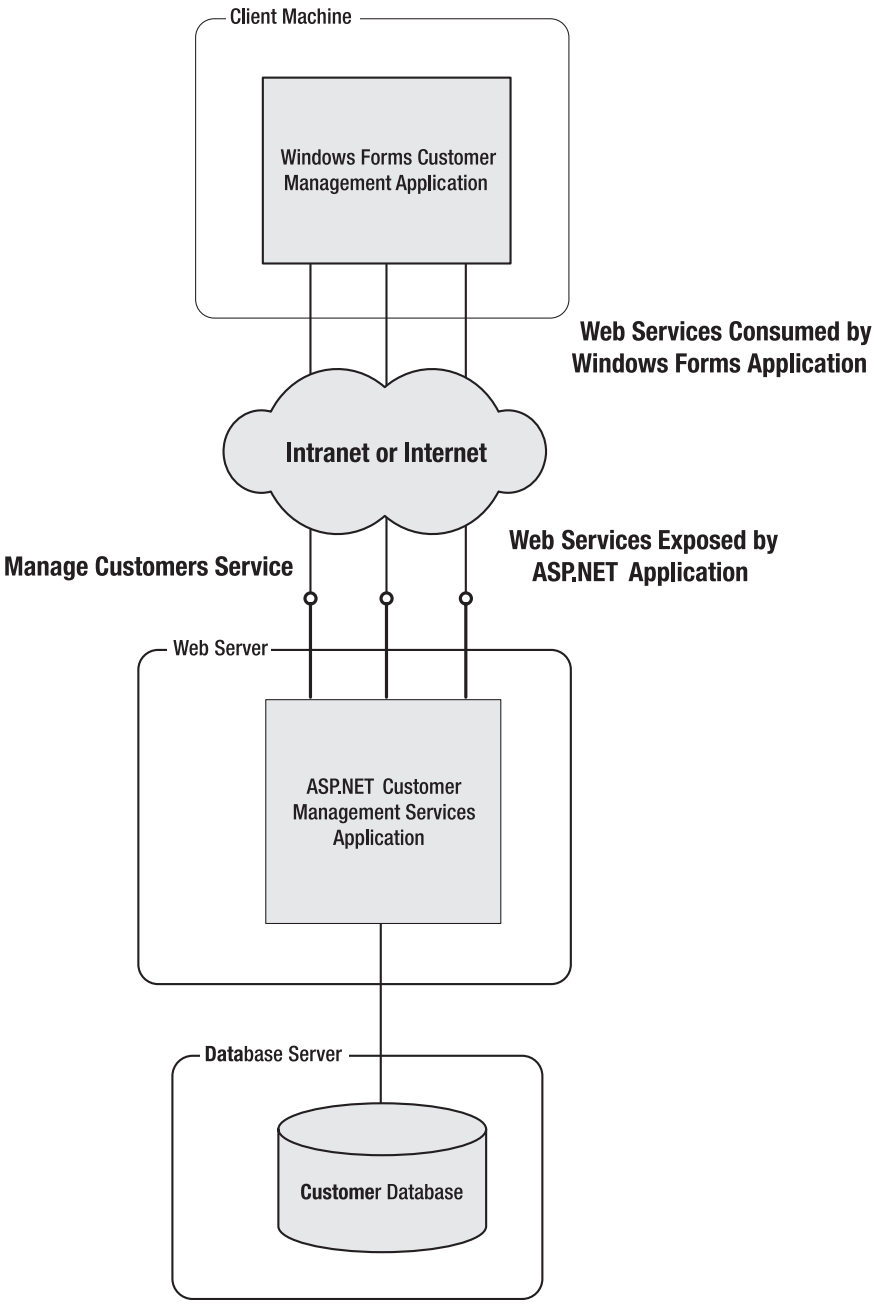

Figure 1-7. *Architecture of an application using smart clients*

Why is it that full-function clients are now acceptable and the DLL Hell described earlier is no longer a problem? The main reason is because .NET assemblies in DLLs are better managed than old-style DLLs. Not only does the global assembly cache (in which .NET assemblies intended to be shared by applications reside) allow for multiple versions of shared assemblies to coexist using the "strong naming" feature, it also introduces the concept of *private assemblies*. When a private assembly needs to be loaded by an executing application, probing takes place in the local folder and any subfolder that is specified by a `.config` file in the local folder. Thus, a whole set of assemblies required only by a particular application may be installed without interfering with those installed for another application. Furthermore, under .NET, unlike COM, there is no longer any need to place details of assemblies in the Registry; it can all be done using copying—known as the "XCopy install."

Let's reconcile these ideas about assembly location with the way that the code creates an object that is an instance of a class defined within an assembly. There are two ways to create an object. The first is when the .NET project references the assembly, which we see clearly in Solution Explorer under the project references. In this case, instances of the class can be created within the code for the referencing project using the `new` keyword. The second way does not require a reference to the assembly to be defined within the project. Instead, the assembly is loaded explicitly by name (and possibly location), and instances of its classes are created using the features of the `System.Reflection` namespace. In either case the probing rules for public or private assemblies apply. This different approach to assemblies makes the installation and upgrade of complex applications much simpler and greatly reduces the management overhead of smart clients compared to old-style thick clients.

There are a couple of other features of smart clients to be considered. One is that they are capable of implementing some sort of disconnected or offline capability. How much capability a client can implement when disconnected from the server depends on the application. It may be possible to produce full functionality, with data synchronization occurring when the client is reconnected, or it may be impossible to provide any functionality at all, other than a polite message indicating disconnection.

The other feature is the ability to perform click-once deployment. Click-once deployment avoids the need to place an executable for an application on each user's machine. Instead, the executable is placed on a common web page from which users can launch it. As part of the launch process, a copy of the application is retrieved to the client's hard drive so that in the future, the application can be launched locally. Applications deployed by the click-once technology can be self-updating and appear in the Start menu and the Add/Remove Programs list. They are deployed in a special location on the client machine, are self-contained, and are kept separate from other installations, but they must be installed separately for each user as they cannot be installed for use by multiple users of the machine.

There are alternatives to click-once deployment. You can use the Windows Installer by packaging all the files needed for deployment into an `.MSI` file, but regardless of whether smart clients are installed by click-once installation or using the Windows Installer, they can always be self-contained and avoid the DLL versioning problems described earlier.

Summary

The goal of this chapter was to introduce Visual Studio 2005 Team System; explain its reason for existence, including the problems it addresses; and give a summary of its features. This chapter also introduced some concepts related to the architecture of enterprise systems; these concepts will be used later in the book. At this point you should

- Understand the purpose of Visual Studio 2005 Team System
- Be familiar with the features it makes available to architects, developers, and testers
- Understand the concept of a distributed application
- Understand the concept of a service-oriented architecture
- Have a picture of the evolution of enterprise system architecture in the past 10 to 15 years and the reasoning behind it
- Be familiar with the way in which smart clients and web clients fit into a service-oriented architecture

We continue our discussion by introducing an imaginary company and describing its project plans. The remainder of this book follows the progress of this project and explains in detail how Visual Studio 2005 Team System is employed throughout.