# 1

# Introducing the Visual Designers



In May 2005, Microsoft published its Visual Studio 2005 Team System Modeling Strategy at `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs05/html/vstsmodel.asp`. In this introductory chapter, we'll draw on the main themes — domain-specific languages (DSLs), model-driven development (MDD), Software Factories, and the Unified Modeling Language (UML) — as they apply to the Visual Studio 2005 Team Edition for Software Architects. We'll summarize what Microsoft has to say on those subjects, and we'll add our own impartial views.

We'll review the evolution of distributed computing architectures — from simple object-oriented development through component and distributed-component design to the service-oriented architectures that represent the current state of the art. That will be our link to the Application Designer that supports a DSL specifically for modeling interconnected Web services applications.

Application Designer is the first in a suite of Distributed System Designers (formerly codenamed "Whitehorse") that also supports system modeling, logical infrastructure modeling, and deployment modeling. We'll introduce each in turn, to be fleshed out in subsequent chapters. To complete the picture in terms of visual design, we'll introduce the Visual Studio 2005 Class Designer, which complements the Distributed System Designers but is not strictly part of that set, and which is available in all Visual Studio editions apart from the Express editions.

Finally, we consider the question "What about UML?" As professional software designers, you may well have been trained in the Unified Modeling Language (UML), the until-now industry standard notation for visual modeling. Like us, you might wonder how — if at all — UML fits into the Visual Studio 2005 Team System scheme; how you can capitalize on your UML skills; and what you do with the UML artifacts you have created.

We'll begin by first establishing the case for undertaking visual modeling — or visual design — at all.

# Why Design Visually?

We'd like to divide that question into two parts: Why design at all, rather than just code? And why design visually?

To answer the first question, we can draw on the common analogy of building complex physical structures, such as bridges. Crossing a small stream requires only a plank of wood — no architect, no workers, and no plans. Building a bridge across a wide river requires a lot more: a set of plans drawn up by an architect so that you can order the right materials; planning the work; communicating the details of the complex structure to the builders, and getting a safety certificate from the local authority. It's the same with software. You can write a small program by diving straight into code, but building a complex software system requires some forethought. You must plan it, communicate it, and document it to gain approval.

The four aims of visual design are therefore as follows:

- To help you visualize a system you want
- To enable you to specify the structure or behavior of a system
- To provide you with a template that guides you in constructing a system
- To document the decisions you have made

Traditionally, design processes like the *Rational Unified Process* have treated design and programming as separate disciplines, at least in terms of tools support. You use a visual modeling tool for design, and a separate IDE for coding. This makes sense if you treat software development like bridge building and assume that the cost of fixing problems during implementation is much higher than the cost of fixing those problems during design. For bridges that is undoubtedly true; but in the realm of software development, is it really more costly to change a line of code than it is to change a design diagram? Moreover, just as bridge designers might want to prototype aspects of their design using real materials, so might software designers want to prototype certain aspects of their design in real code. For these reasons, the trend has been toward tools that enable visual design and coding within the same environment, with easy switching between the two representations, thus treating design and coding as essentially two views of the same activity. The precedent was set originally in the Java space by tools such as Together-J, and more recently in the .NET space by IBM-Rational XDE, and this approach has been embraced fully by the Team Edition for Software Architects.

Now the second part of the question: If the pictorial design view and the code view are alternative but equivalent representations, then why design visually at all? The answer to the second part of the question is simple: A picture paints a thousand words. To test that theory, just look at the figures in this chapter and imagine what the same information would look like in code. Then imagine trying to explain the information to someone else using nothing but a code listing.

Of course, if we're going to use visual modeling as a communication tool, the notation must be commonly understood. In UML terms, this means an industry-standard, general-purpose notation that is all things to all people. In Visual Studio 2005 terms, this means a set of domain-specific notations that are highly tuned for the information they are designed to convey. Nonetheless, the first version notations are familiar enough to be immediately accessible to most UML-trained software designers.

# Microsoft's Modeling Strategy

As we stated at the outset, Microsoft's Visual Studio 2005 Team System Modeling Strategy is based on three key ideas: domain-specific languages (DSLs), model-driven development (MDD), and Software Factories.

We consider these three topics together as comprising Microsoft's new vision for how to add value to the software development process through visual modeling. It's this "adding value" that distinguishes the new vision from the old vision, which — to put a label on it — we'll call UML.

First to set the scene: The Object Management Group (OMG) has a licensed brand called *Model-Driven Architecture (MDA)*. MDA is an approach to MDD based on constructing platform-independent UML models (PIMs) supplemented with one or more platform-specific models (PSMs). Microsoft also has an approach to model-driven development, based not on the generic UML but on a set of tightly focused domain-specific languages (DSLs). This approach to model-driven development is part of a Microsoft initiative called *Software Factories,* which in turn is part of a wider Dynamic Systems Initiative.

> *If you would like a more in-depth exploration of software factories, we recommend that you pick up* Software Factories: Assembling Applications with Patterns, Works, Models and Tools, *written by Keith Short and Jack Greenfield (Wiley & Sons Publishing; ISBN: 0471202843).*

## *Model-driven development*

As a software designer, you may be familiar with the "code-generation" features provided by UML tools such as Rational Rose and IBM-Rational XDE. These tools typically do not generate "code" at all but merely "skeleton code" for the classes you devise, so all you get is one or more source files containing classes populated with the attributes and operation signatures that you specified in the model.

> *The words "attribute" and "operation" are UML terminology. In the .NET world, we tend to refer to these as "field" and "method," respectively.*

As stated in Microsoft's modeling strategy, this leads to a problem:

"If the models they supported were used to generate code, they typically got out of sync once the developers added other code around the generated code. Even products that did a good job of 'round tripping' the generated code eventually overwhelmed developers with the complexity of solving this problem. Often these problems were exacerbated because CASE tools tried to operate at too high a level of abstraction relative to the implementation platform beneath. This forced them to generate large amounts of code, making it even harder to solve the problems caused by mixing hand-written and generated code."

The methods that are generated for each class by UML code generation tools typically have complete signatures but empty bodies. This seems reasonable enough because, after all, the tool is not psychic. How would it know how you intend to implement those methods? Well, actually, it could know.

UML practitioners spend hours constructing dynamic models such as statecharts and sequence diagrams that show how objects react (to method invocations) and interact (invocate methods on other objects). Yet that information, which could be incorporated into the empty method bodies, is lost completely during code generation.

*We should point out that not all tools lose this kind of information during code generation, but most of the popular ones do. In addition, in some cases UML tools do generate code within method bodies — for example, when you apply patterns using IBM-Rational XDE — but in general our point is valid.*

Why do UML tools generally not take account of the full set of models during code generation? In part it's because software designers do not provide information in the other models with sufficient precision to be useful as auto-generated method bodies. The main reason for that is because the notation (UML) and tools simply do not allow for the required level of precision.

What does this have to do with model-driven development? Well, MDD is all about getting maximum value out of the modeling effort, by taking as much information as possible from the various models right through to implementation. As Microsoft puts it:

"Our vision is to change the way developers perceive the value of modeling. To shift their perception that modeling is a marginally useful activity that precedes real development, to recognition that modeling is an important mainstream development task. . . "

Although our example of UML dynamic modeling information finding its way into implemented method bodies was useful in setting the scene, don't assume that model-driven development is only, or necessarily, about dynamic modeling. If you've ever constructed a UML deployment model and then tried to do something useful with it — such as generate a deployment script or evaluate your deployment against the proposed logical infrastructure — you will have seen how wasted that effort has been other than to generate some documentation.

The model-driven development ethos translates into Visual Studio 2005 functionality as follows:

- Automated validation of application settings against constraints of the hosting environment
- Generation of Web services wrappers to bridge different implementation technologies
- Automatic synchronization between source code and class models (therefore, no troublesome code generation and reverse engineering phases)
- Production of deployment reports in support of automated deployment via external scripting tools

The bottom line? Well, because models are regarded as first-class development artifacts, developers write less conventional code and development is therefore more productive and agile. In addition, it fosters a perception among all participants — developers, designers, analysts, architects, and operations staff — that modeling actually *adds value* to their efforts.

## Domain-specific languages

UML fails to provide the kind of high-fidelity domain-specific modeling capabilities required by automated development. In other words, if you want to automate the mundane aspects of software development, then a one-size-fits-all generic visual modeling notation will not suffice. What you need is one or more domain-specific languages (or notations) highly tuned for the task at hand — whether that task is the definition of Web services, the modeling of a hosting environment, or traditional object design.

**A domain-specific language (DSL) is a modeling language that meets certain criteria. For example, a modeling language for developing Web services should contain concepts such as Web methods and protocols. The modeling language should also use meaningful names for concepts, such as fields and methods (for C#) rather than attributes and operations. The names should be drawn from the natural vocabulary of the domain.**

The DSL idea is not new, and you may already be using a DSL for database manipulation (it's called SQL) or XML schema definition (it's called XSD).

Visual Studio 2005 embraces this idea by providing domain-specific languages for specific tasks. Domain-specific languages enable visual models to be used not only for creating design documentation, but also for capturing information in a precise form that can be processed easily, raising the prospect of compiling models into code.

In the section labeled "Visual Designers" later in this chapter, we'll introduce the initial set of DSLs that Microsoft has devised for service-based application modeling, logical infrastructure modeling, system modeling, and deployment modeling. But that's not an exhaustive set. We can expect additional visual designers to be incorporated into future versions of Visual Studio — perhaps to support business process modeling or Web-services contract design. In addition, Microsoft has a suite of tools that enables you to devise your own domain-specific languages for your own problem domain.

In that context, "your own problem domain" need not be technology focused, such as how to model Web services or deployment infrastructures, but may instead be business focused. You could devise a DSL that is highly tuned for describing banking systems or industrial processes. And here's an interesting idea: If you really can't live without UML, how about devising a set of DSLs that mirrors the UML?

We'll say more about that at the end of the chapter, and you can also find out more about DSLs in Chapter 7.

## Software Factories

In their book *Software Factories* (Wiley, 2004) Greenfield, Short, and others introduce and discuss techniques and tools for adding significant value to the process of turning visual models into functional implementations.

In a nutshell, the Software Factories initiative is all about pulling together the ideas of model-driven development and domain-specific languages to support the following:

- Full or partial generation of artifacts (such as source code and configuration files) from other artifacts (particularly models)
- Synchronization of related artifacts (such as the model and the code) during development, and validation of artifacts constructed manually
- The application of patterns and industry best practices *driven by guidance in context.*

The key to all that is a Software Factory Schema, which relates work done at one level of abstraction, in one part of the system or in one phase of the life cycle to work done at other levels or in other parts and phases.

While some of the results of this initiative are already apparent in the Visual Studio 2005 Team System, some of the ideas have yet to be fully realized in tools, and this is not just about Microsoft tools. To quote from Microsoft's published modeling strategy:

". . . we see factories as the basis of a broad ecosystem in which our customers and partners participate, building custom factories on top of foundations we supply, and supplying factory components to other members of the ecosystem."

A concrete example would be a software factory for *banking*. This would include DSL-based modeling tools, process guidance, and architectural frameworks that help to automate the tasks involved in building systems specifically for the banking industry; the point being that all banking systems comprise the same sorts of entities (accounts, interest rates, etc.), the same kinds of processes (bill payments, funds transfers, etc.), as well as being bound by the same regulatory requirements. It makes little sense to reinvent the wheel for each new process.

> **For more information on this initiative, visit the Software Factories Workbench at**
> `http://lab.msdn.microsoft.com/teamsystem/workshop/sf/`.

# From Objects to Services

The design features provided by Visual Studio 2005 Team Edition for Software Architects have been influenced not only by Microsoft's vision for model-driven development, but also by a technological evolution from object-based architectures through (distributed) component-based architectures to the service-oriented architectures (SOA) that represent the current best practice in distributed system design.

This section summarizes that evolution — in part to demonstrate that service orientation is a good idea, and in part as a natural lead-in to the first of the new visual designers, the Application Designer, which provides a DSL specifically for modeling interconnected service-based applications.

## *Objects and compile-time reuse*

When object-oriented programming became popular in the mid-1990s, it was perceived as a panacea. In theory, by combining state (data) and behavior (functions) in a single code unit, we would have a perfectly reusable element: a cog to be used in a variety of machines.

The benefit was clear. No more searching through thousands of lines of code to find every snippet that manipulated a date — remember the Y2K problem? By encapsulating all date manipulation functionality in a single `Date` class, we would be able to solve such problems at a stroke.

Object orientation turned out not to be a panacea after all, for many reasons, including — but not limited to — bad project management (too-high expectations), poor programming (writing procedural code dressed up with objects), and inherent weaknesses in the approach (such as tight coupling between objects).

For the purposes of this discussion, we'll concentrate on one problem in particular, which is the style of reuse that objects encouraged — what you might call *copy-and-paste reuse*.

Consider the following copy-and-paste reuse scenario: You discover that your colleague has coded an object — call it `Book` — that supports exactly the functionality you need in your application. You copy the entire source code for that object and paste it into your application.

Yes, it has saved you some time in the short term, but now look a little further into the future.

Suppose the `Book` class holds fields for `Title` and `ISBN`, but in your application you now need to record the author. You add a new field into your copy of the `Book` source code, and name that field `Author`.

In the meantime, your colleague has established the same need in his application, so he too modifies the `Book` source code (his copy) and has the foresight to record the author's name using two fields: `AuthorSurname` and `AuthorFirstname`.

Now the single, reusable `Book` object exists in two variants, both of which are available for a third colleague to reuse. To make matters worse, those two variants are actually incompatible and cannot easily be merged, thanks to the differing representations of the author name.

Once you've compiled your application, you end up with a single executable file (.exe) from which the `Book` class is indivisible, so you can't change the behavior of the `Book` class — or substitute it for your colleague's variant — without recompiling the entire application (if you still have the source code, that is!).

As another example (which we'll continue through the next sections), imagine you're writing a technical report within your company. You see one of the key topics written up in someone else's report, which has been sent to you by e-mail. You copy their text into your document, change it a little, and now your company has two — slightly different — descriptions of the same topic in two separate reports.

## Components and deploy-time reuse

At this point you might be shouting that individual classes could be compiled separately and then linked together into an application. Without the complete source code for the application, you could recode and replace an individual class without a full recompilation, just link in the new version.

Even better, how about compiling closely related (tightly coupled) classes into a single unit with only a few of those classes exposed to the outside world through well-defined interfaces? Now the entire sub-unit — let's call it a *component* — may be replaced with a newer version with which the application may be relinked and redeployed.

Better still, imagine that the individual components need not be linked together prior to deployment but may be linked on-the-fly when the application is run. No need to redeploy the entire application then; just apply the component updates. In technological terms, we're talking here about DLLs (for those with a Microsoft background) or JAR files (for the Java folks). And in .NET terms, we're talking about assemblies.

Continuing our nonprogramming analogy, consider hyperlinking your technical report to the appropriate section of your colleague's report and then distributing the two documents together, rather than copying his text into your document.

## Distributed components and run-time reuse

Continuing with this line of thought, imagine that the components need not be redeployed on client devices at all. They are somehow just available on servers, to be invoked remotely when needed at runtime.

In our nonprogramming example, consider not having to distribute your colleague's report along with your own. In your own report, you would simply hyperlink to the relevant section in your colleague's document, which would be stored — and would remain — on an intranet server accessible to all recipients.

One benefit of this kind of remote linking is that the remote component may be adapted without having to be redeployed to numerous clients. Clients would automatically see the new improved version, and clients constrained by memory, processing power, or bandwidth need not host the components locally at all.

This leads us to a distributed component architecture, which in technology terms means DCOM, CORBA, or EJB. All of these technologies support the idea of a component — or object — bus via which remote operations may be discovered and invoked. In Figure 1-1, the component bus is indicated by the grayed-out vertical bar.

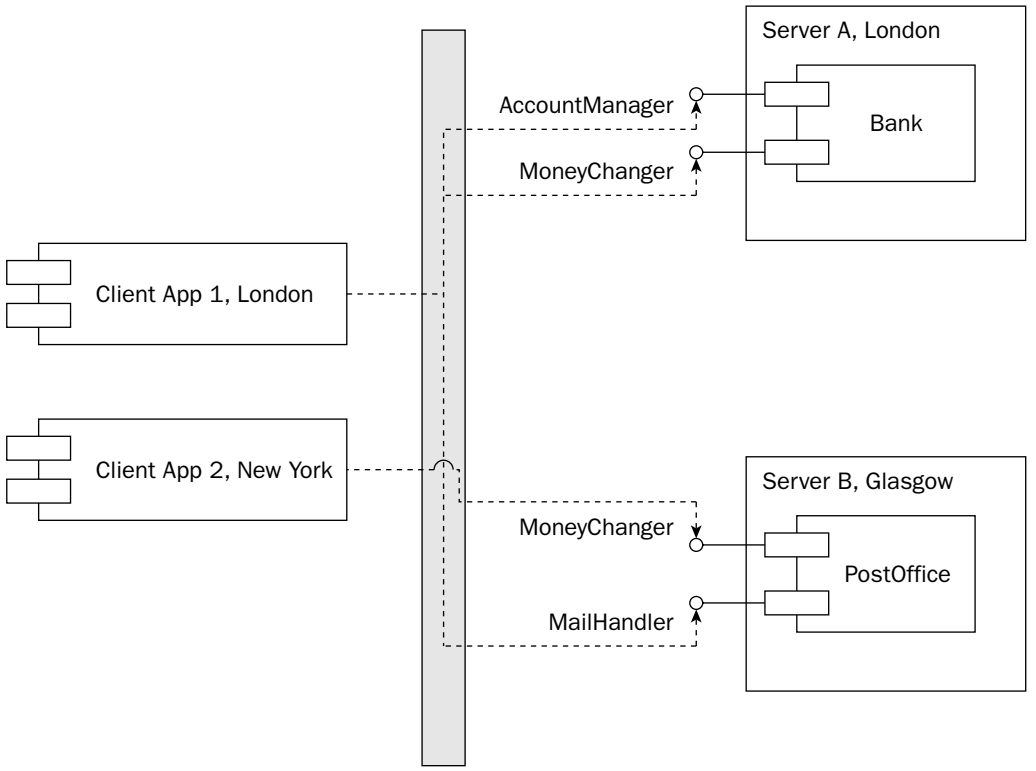


**Figure 1-1**

*In case you're wondering, the notation used in Figure 1-1 is UML. In fact, it's a UML Component Diagram drawn using Visio for Enterprise Architects. We used UML here because we haven't yet introduced you to the new notations.*

Of course, as remote components are modified, we must ensure that none of the modifications affect clients' abilities to use those components, which is why we must make a distinction between interfaces and implementations:

- **Interface:** A component interface defines the contract between that component and the clients that use it. This contract must never be broken, which in practice means that existing operations may not have parameters added or taken away, though it may sometimes be permissible to add new operations.
- **Implementation:** A component implementation may be changed at will in terms of the use of an underlying database, the algorithms used (e.g., for sorting), and maybe even the programming language in which the component is written, as long as the behavior is unaffected as far as the client is concerned.

This distinction between interface and implementation raises some very interesting possibilities. For example, a single component implementation (e.g., `Bank`) may support several interfaces (e.g., `AccountManager` and `MoneyChanger`) whereas the `MoneyChanger` interface may also be supported by another implementation (e.g., `PostOffice`).

Moreover, the underlying implementations may be provided by various competing organizations. For example, you could choose your bank account to be managed by the Bank of BigCity or the National Enterprise Bank so long as both supported the `AccountManager` interface. This idea is revisited in Chapter 2.

## *Distributed services and the service-oriented architecture*

What's wrong with the distributed components approach?

To start with, the same underlying concepts have been implemented using at least three different technologies: the OMG's Common Object Request Broker Architecture (CORBA), Microsoft's Distributed Component Object Model (DCOM), and Sun Microsystems' Enterprise Java Beans (EJB). Though comparable in theory, these approaches require different programming skills in practice and do not easily interoperate without additional bridging software such as DCOM/CORBA bridges and RMI-over-IIOP.

Furthermore, distributed component technologies encourage stateful intercourse between components by attempting to extend the full object-oriented paradigm across process and machine boundaries, thereby triggering a new set of challenges such as how to manage distributed transactions across objects, requiring yet more complex technology in the form of the CORBA Transaction Service or the Microsoft Transaction Server (MTS).

To a certain extent, a service-oriented architecture alleviates these problems by keeping it simple: by making the services stateless if possible, and by allowing services to be invoked using the widely adopted, standard over-the-wire protocol Simple Object Access Protocol (SOAP).

Logically, a representation of Figure 1-1 (distributed component bus) redrawn as Web services would be virtually identical, as shown in Figure 1-2 (distributed Web services). Admittedly, this diagram has been drawn specifically to look as much like the other one as possible; nevertheless, you should note the close correspondence between the various elements.

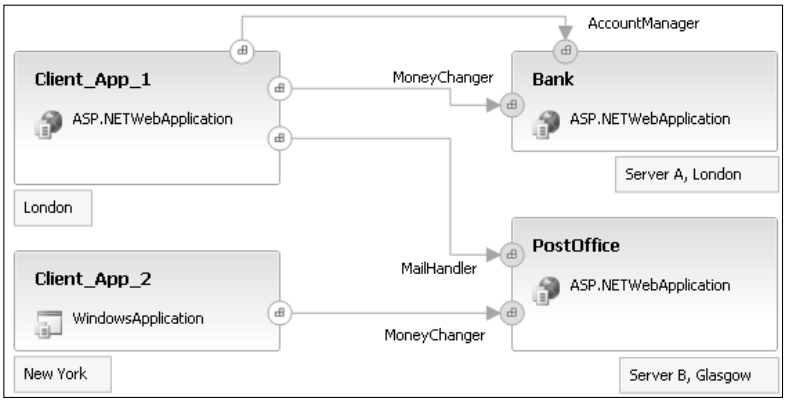


Figure 1-2

Not only does Figure 1-2 show the close correspondence — and logical progression — from distributed components to Web services, it also offers a first glimpse at the kinds of diagrams you'll be drawing in this part of the book. It is a Visual Studio 2005 application diagram drawn using Application Designer.

Before we move on, it's important to understand a key point here: Just because the original object-oriented paradigm has evolved toward the service-oriented architecture, and just because Application Designer (described in the next section) is biased toward service-oriented architectures, that doesn't mean that the earlier object-oriented and component-based approaches have been replaced — they are merely complemented. In fact, in Chapter 5, you'll do some traditional object-oriented design using the Visual Studio 2005 Class Designer.

# Visual Designers

Visual design within Visual Studio is not new, hence the name "Visual" Studio. For some time there have been visual designers for Windows forms and web forms so that you can lay out your screen designs without resorting to code, and so that you can add behavior to buttons and other controls without having to remember how to code their event handlers. There are also other visual designers such as the XML schema designer.

The new suite of visual designers complements the existing set by providing capabilities for modeling the static aspects of your application architecture, deployment infrastructure, and lower-level object design. The complete list of new designers that we'll be introducing in this chapter and covering in-depth in this part of the book is as follows:

- Application Designer
- Logical Datacenter Designer
- System Designer
- Deployment Designer
- Class Designer

As stated earlier, the first four comprise the Distributed System Designers (formerly called "Whitehorse") that are unique to the Visual Studio 2005 Team Edition for Software Architects. Class Designer is not unique to that edition, so you'll find it also in the Team Developer and Team Tester editions. We treat it as a member of the same set — rather than relegate it to another part of the book — because it operates in a similar fashion and contributes to the same overall visual modeling experience.

Each is covered in turn, although System Designer and Deployment Designer are covered together because they are intrinsically linked. Although you can use Application Designer and Logical Datacenter Designer individually, it makes little sense to use Deployment Designer without System Designer.

> *In fact, all of the designers are interrelated in very important ways via the common SDM format, as you'll see later. All we're saying here is that you can — and in fact, should — draw an Application Diagram separately from a Logical Datacenter Diagram, but it makes much less sense (arguably it's not possible) to draw a Deployment Diagram without the benefit of a set of System Definitions.*

Note that the existing designers for forms and XML schemas have a direct correspondence with the underlying code. This is true also of the new designers, which, unlike their UML equivalents, are not merely abstract representations of the underlying code.

As we introduce each of the visual designers, keep in mind that we're doing just that — introducing them. Our aim is to provide only a preview of each of the diagram types, and to place those diagrams relative to one another in the Software Development Lifecycle (SDLC). As it happens, the diagrams that we use in the following preview are all taken from the StockBroker case study that runs through the subsequent chapters, but we're not setting out to explain the details of that running example in this chapter. You'll be introduced to it more formally at the end, as a road map for the chapters that follow.

## Application Designer

Application Designer enables you to define the major applications (like components in UML), their endpoints (like interfaces in UML), and their interconnections. An example application diagram based on the StockBroker case study that we'll introduce later is given in Figure 1-3.

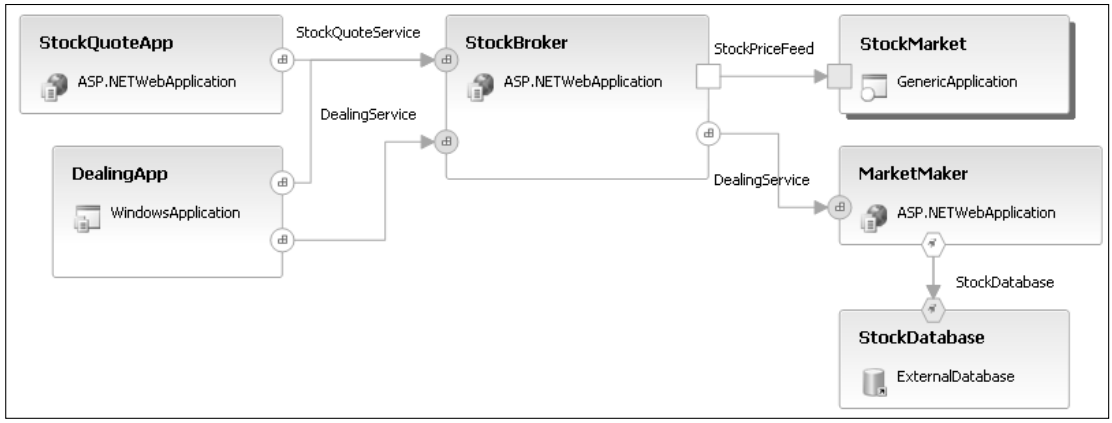


Figure 1-3

You can populate such diagrams by dragging application prototypes from the toolbox onto the diagram. These prototypes, rendered as stylized box shapes, represent ASP.NET web applications, Web services, Windows applications, and other application types. Once placed, the applications are adorned with consumer and supplier endpoints between which connections may be made.

You are allowed only one Application Diagram per solution, because ultimately it defines how your solution is composed of projects. If you implement this diagram (discussed in Chapter 2), you end up with a separate project for each application shown on the diagram, except for the MarketMaker and StockDatabase applications, which cannot be implemented.

> *In Chapter 2, you'll see how some applications may be implemented in a solution, whereas some — such as generic applications, external Web services, and databases — may not.*

It's no accident that we discussed service-oriented architecture (SOA) immediately before we introduced this designer, because it is biased very much toward designing service-oriented applications. While you'll find good support for adorning applications with Web service endpoints, you'll find no out-of-the-box support for adding .NET Remoting endpoints or COM endpoints as alternative communication mechanisms.

## *Logical Datacenter Designer*

Logical Datacenter Designer enables you to define one or more deployment architectures in terms of communication boundaries, logical servers, and the protocols that interconnect them (see Figure 1-4 for an example taken from our StockBroker case study).

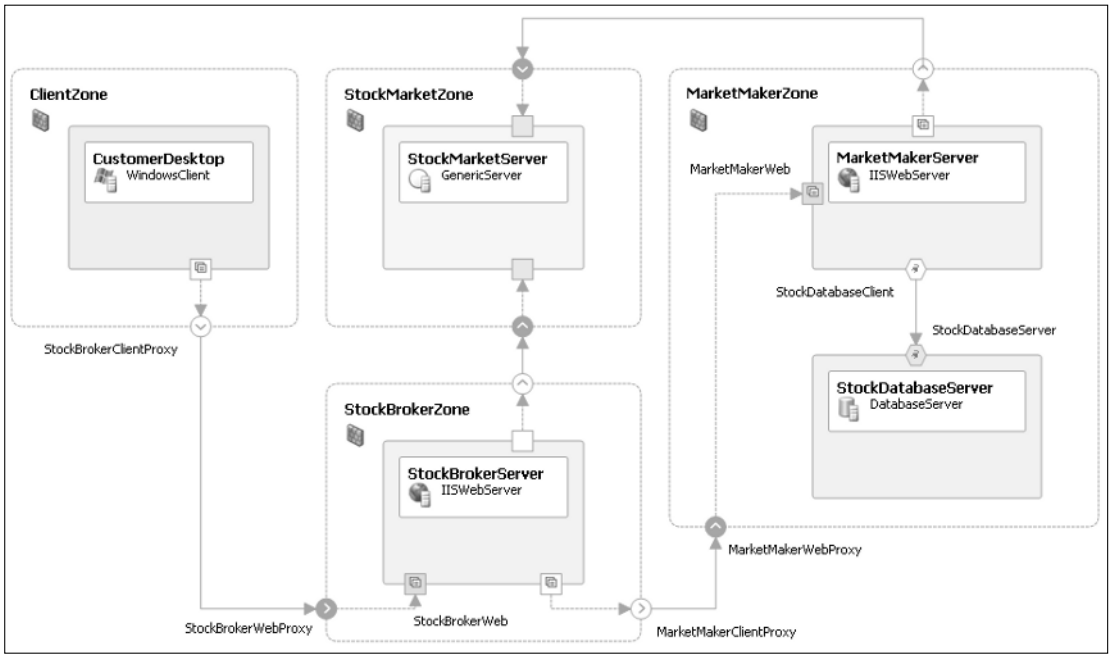


Figure 1-4

You populate such diagrams by dragging zone and server prototypes from the toolbox onto the diagram. Zones are rendered as dashed boxes representing protected network regions, and logical servers are rendered as stylized boxes representing deployment hosts. Zones and servers may be adorned with endpoints between which connections may be made.

Note that that this designer is used for *logical* infrastructure modeling, not *physical* infrastructure modeling. The five servers shown in the diagram could be hosted on five separate machines, across four machines — one in each zone — or even all on one machine, perhaps running in Virtual PCs.

You can have as many logical datacenter diagrams as you like, and the diagrams may be moved freely between solutions.

## System Designer and Deployment Designer

We cover System Designer and Deployment Designer together because it makes little sense to show you one without the other. In fact, even in the simplest case of defining a default deployment scenario straight from an application diagram (as you'll see in Chapter 2), a system is created implicitly.

The two design diagrams described so far provide two alternate views of the overall architecture. The application diagram represents a solution-scoped view of the applications that you have designed, and the logical datacenter diagram represents the operations analyst's view of the logical infrastructure on which the system will be deployed. A deployment diagram depicts a deployment of the applications in a system to the logical servers in a logical datacenter. In essence, the System Designer and Deployment Designer serve to tie the two views together.

By default, each application from the application diagram binds onto an individual deployable system, but using System Designer it is possible to specify an alternative binding — for example, two applications can be combined into a single deployable system. Figure 1-5 shows such a system, along with the System View window from which you can select the applications for each system.

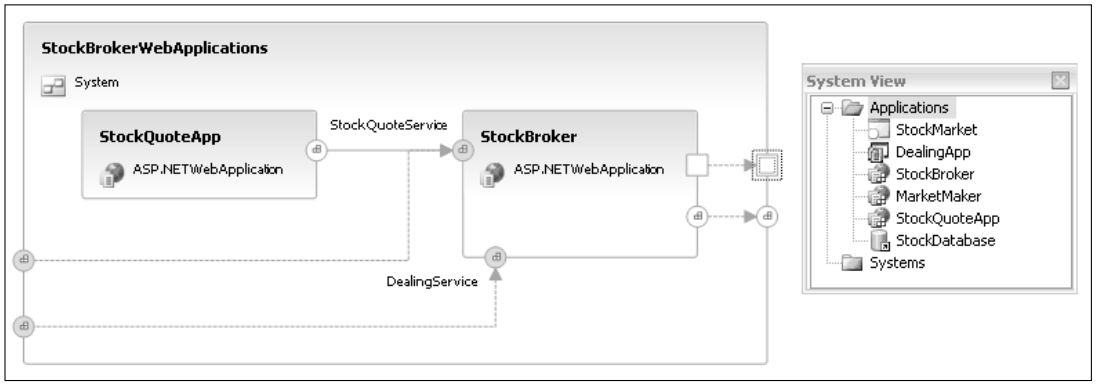


Figure 1-5

At first glance, the design surface of Deployment Designer (see Figure 1-6) looks very much the same as Logical Datacenter Designer. Look closer, however, and you'll see that each of the servers has one or more applications (also taken from the System View) bound onto it.

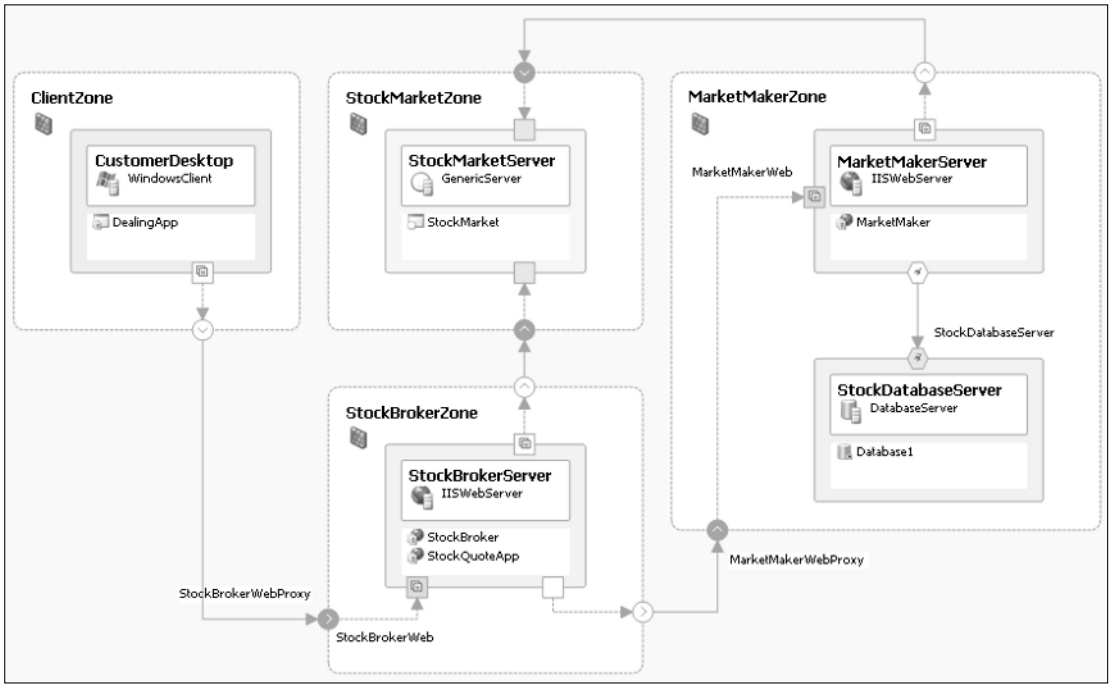


Figure 1-6

## Class Designer

If you mention visual modeling to most software developers, the first thing that will spring into their minds is the UML class diagram that shows how object classes are connected together in a persistent sense or to allow navigation between them. Visual Studio 2005 provides a Class Designer most closely resembling its UML namesake.

> *Class Designer is not limited to the Visual Studio 2005 Team Edition for Software Architects. You can do class modeling in the Team Developer and Team Tester editions too.*

Figure 1-7 provides an example Class Designer class diagram showing classes (with fields and methods), interfaces (shown as lollipops), associations, and inheritance.

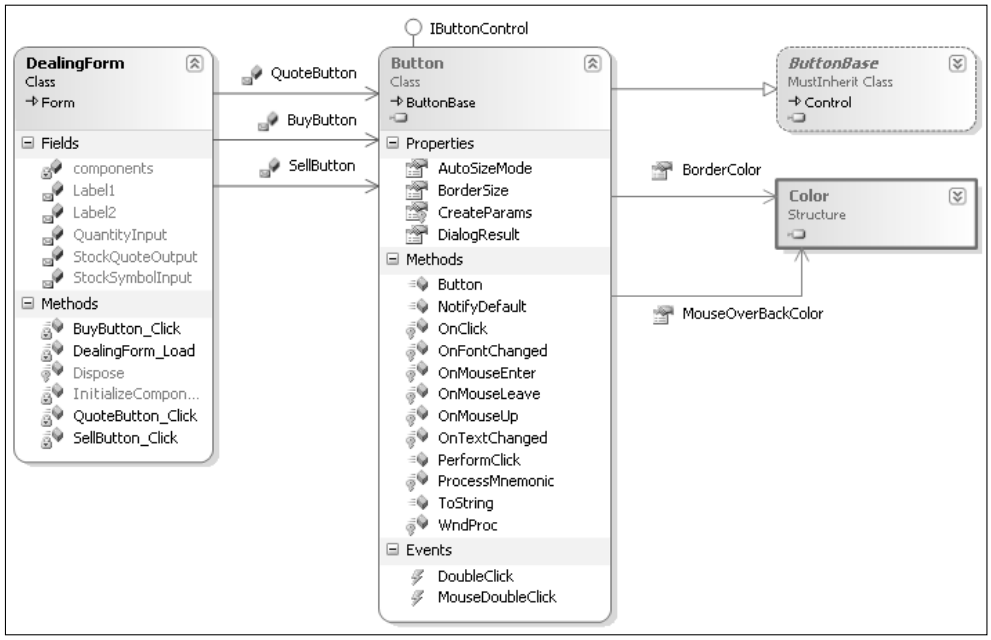


Figure 1-7

## Code synchronization

Historically, visual design tools such as Rational Rose were distinct tools, separate from the development tools used for coding. Integration was achieved via a code-generation facility that would produce a one-shot cut of skeleton code as source files that could be loaded up in the IDE. Conversely, existing source files and compiled class libraries could be reverse-engineered into the design tool to document existing code using UML notation.

Some tools — including Rose but excluding Visio EA — provided a degree of round-trip engineering whereby code could be generated from the model in the design tool, modified in the IDE, reverse-engineered back into the model, and so on, ad infinitum. This round-trip engineering was not always as effective as the tool vendor claimed for the following reasons:

- It was not always possible for the tool to reconcile every change made outside of the tool. For example, a class renamed in code is indistinguishable — as far as the tool is concerned — from a class being deleted and a new one being created.
- Synchronization issues could arise in team scenarios, such as multiple designers and developers modifying portions of the model and the code in incompatible ways, resulting in an intractable merge.

A new precedent was set by IBM-Rational XDE, which keeps the two views — model and code — perfectly synchronized with no need to explicitly generate code or reverse engineer. Any changes you make in code are automatically reflected in the model, and vice-versa.

That is the approach taken by Class Designer, which means you won't find any feature labeled "generate code" or "reverse engineer." They're simply not needed. We'll discuss this further in Chapter 5, but for now the main implications of this are as follows:

- Unlike UML, which can be rather generic in its class modeling offering, Class Designer uses the data types provided by the underlying implementation language, whether Visual Basic, Visual C#, or Visual J#. The Visual Studio 2005 Class Designer does not support Visual C++.
- The three kinds of association defined by UML — ordinary association, aggregation, and composition — are not distinguished on diagrams because they cannot be distinguished in code.
- Class Designer is just as useful for visualizing existing classes and other types as it is for defining new ones.

# Introducing the StockBroker Case Study

A single example, the StockBroker case study, will be used as a consistent thread running throughout the chapters in this part of the book. We regard the StockBroker scenario as a fair representation of the kind of service-oriented distributed system that you might encounter in the real world, and toward which the Distributed System Designers are targeted. In this context, an example based on the usual personal CD cataloging program simply wouldn't do justice to the tools.

If you refer to our summary of the visual designers previously discussed, you'll notice that the diagrams included items with names such as StockBroker, MarketMaker, DealingApp, and StockDatabase, which are the essential items from which the case study is composed. In the following chapters, you'll meet those figures again in the following contexts:

- In Chapter 2, you'll learn how to use Application Designer to sketch out the overall structure of the StockBroker example in terms of applications and the connections between them. You'll also learn how to generate and locally deploy implementation code directly from that design.
- In Chapter 3, we'll define a logical datacenter using Logical Datacenter Designer to host the case study example, independent of the application design devised in Chapter 2.
- Chapter 4 pulls the two together by showing how a combination of System Designer and Deployment Designer may be used to bind the applications from the application diagram onto the datacenter defined by the LDD.
- In Chapter 5, you'll revisit the implementation code from Chapter 2 as a vehicle for learning how Class Designer may be used to model the internal implementations of each application.

Though a single case study example is used for consistency, and to give credibility to the overall process, we will step outside of that example as necessary to demonstrate concepts and mechanisms that do not fit neatly into that scheme.

# Designer Relationships and Team System Integration

Being presented with five new visual designers, each representing a different view of the overall system, you might be wondering how they all fit together. This section describes the important relationships between the designers, in the context of who uses which designer.

> *Traditional methods of teaching visual modeling often present a set of notations that individually represent one aspect — or view — of the system without describing how those various views are interrelated. You might learn, for example, that a UML statechart shows transitions between states triggered by events; and that a UML sequence diagram shows messages being passed between objects; but without further guidance, how do you deduce that the events in the statechart are related to the messages in the sequence diagram?*

In Figure 1-8, we have used a Visio UML Activity Diagram for convenience to show how various roles within your software development organization would typically use the visual designers, including the artifacts that would be produced or consumed at each stage.

> *We've not used a Team System diagram because as yet there is no diagram suited to that purpose. That situation might change with future releases; in the meantime, we see a retained role for Visio in cases such as this.*

You can read the diagram shown in Figure 1-8 in one of two ways:

- A *process flow diagram* (in bold) showing the activities performed by the various roles — that is, who uses which of the visual designers
- An *object flow diagram* (not bold) showing the objects produced and consumed by each activity — that is, the artifacts exchanged between the design tools

The following sections describe each of those perspectives in turn.

## *Process flow (roles and activities)*

The role names that we have used are only indicative, because in reality the exact role names and the scopes of those roles will depend on the process that you follow: MSF for Agile Software Development, MSF for CMMI Process Improvement, the Unified Process, or whatever else you choose. Regardless of process, the order in which the designers are used and their interactions via artifacts are likely to be as shown in Figure 1-8.

Initially, the process flow branches into two parallel activities such that operations analysts may use Logical Datacenter Designer to define the logical infrastructure, and the application solution designer/architect may use Application Designer to define the overall shape of the system, completely independently.

Once the Application Design has been completed, the designer/developer can define the code structure for each application using Class Designer.

Once the Application Design and the Logical Datacenter Design have been completed, the application solution designer/architect can use System Designer to group applications into systems, and then use Deployment Designer to specify and evaluate how the application systems will be deployed on servers of the logical datacenter.

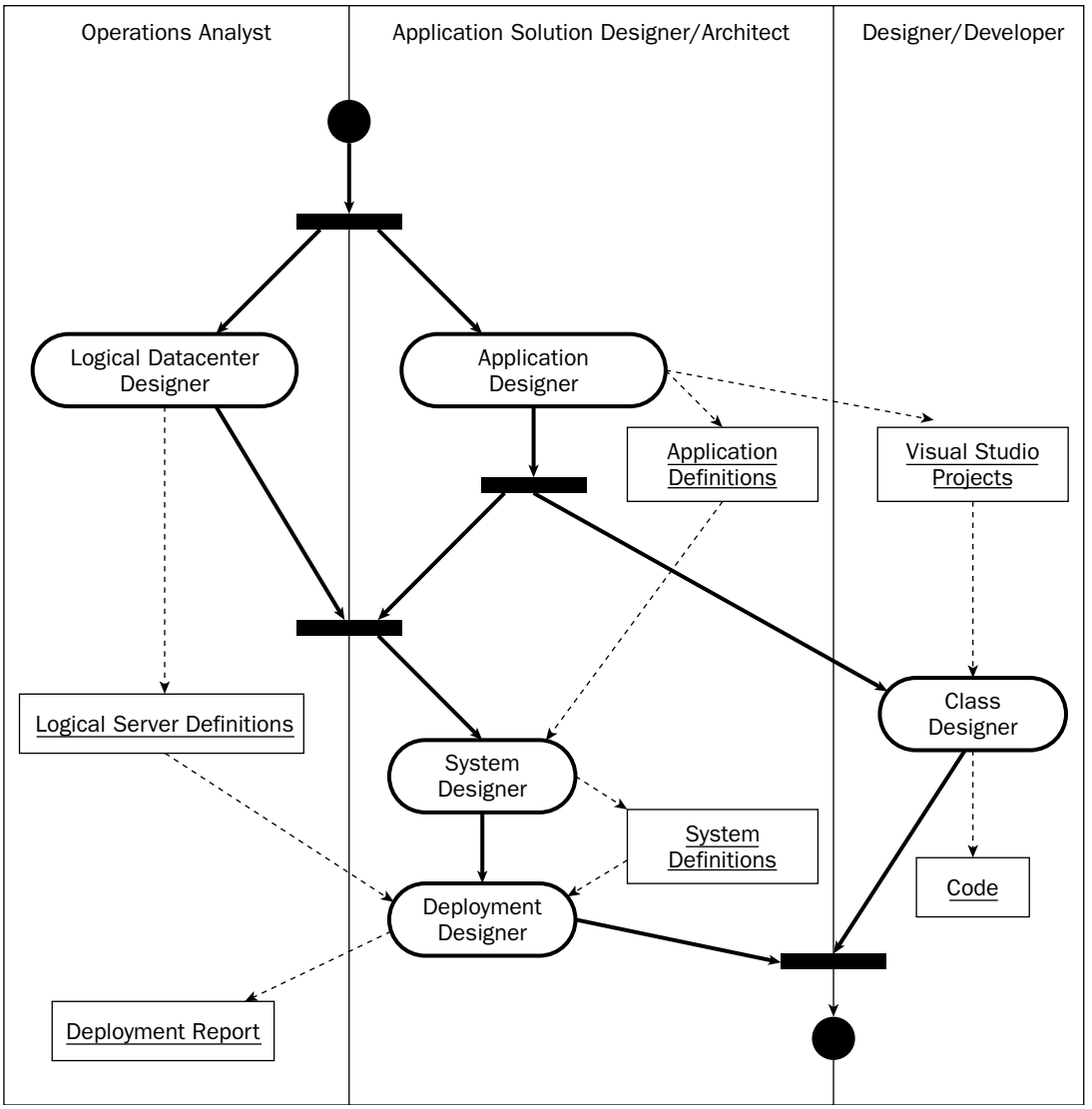


Figure 1-8

## *Object flow (artifacts)*

The output from Application Designer is a set of Application Definitions expressed in SDM syntax and stored in an .ad file. If you generate projects, the SDM files that describe the applications are stored in the projects as .sdm files.

Class Designer may be used to refine and restructure the skeleton projects code and to define new code structures within those projects, resulting in final deliverable code.

System Designer accepts Application Definitions and enables you to compose systems as groups of applications, resulting in a set of System Definitions stored in an .sd file and again expressed in SDM syntax.

The output from Logical Datacenter Designer is a set of Logical Server Definitions expressed in SDM syntax and stored in an `.ldd` file.

Deployment Designer accepts System Definitions and enables you to map systems onto servers, resulting in a deployment map (as SDM) and a Deployment Report. The Deployment Report may be human-readable (as HTML) or machine-readable (as XML). The latter allows for automated deployment via scripting.

## Settings and constraints

When you produce application definitions using Application Designer, you can specify settings and constraints for those applications, and these may be refined when you produce system definitions using System Designer. You can also specify settings and constraints when you produce logical server definitions using Logical Datacenter Designer.

What you need to understand is that the settings and constraints defined using Logical Datacenter Designer work in the opposite manner to those defined using Application Designer and System Designer. Application and System settings must be compatible with the constraints of the Logical Server that will host them. Logical server settings must be compatible with constraints of the applications and systems to be hosted.

By defining constraints on a server, the operations analyst is saying to the application architect, "If you want to host your application on this server of my logical datacenter, then you must satisfy these constraints via your settings." By defining constraints on an application or system, the application architect is saying to the operations analyst, "To host my application, you need to provide me with a server whose settings match the constraints of my application."

## Source control and item tracking

Because the diagrams that you draw, and the application and server definitions that you produce, not to mention the project source code, are all stored in files, these may be checked in or out of the source control portion of the Team System and tracked as artifacts, in part to eliminate clashes during team working (i.e., two people trying to work on the same diagram at the same time) and in part to provide a versioned history of certain artifacts. For example, the operations team might want to baseline the logical datacenter diagram at various stages, by version controlling the `.ldd` files.

When controlling diagrams and files, you need to consider whether those items have an independent life cycle or whether they are intrinsically linked to specific Visual Studio solutions. For example, the `.ldd` files that define a logical datacenter are truly portable. Therefore, they may be included in any number of solutions or many can be contained within one solution. Conversely, the `.ad` files that define the application design are limited to one per solution, and tie into a specific solution once implemented. You can have as many class diagrams in a project as you like, but these are tied in absolutely to the source code that they visualize.

# Introducing the System Definition Model

It's doubtful whether anyone would attempt to design an application these days without the aid of a modeling tool, using nothing but pencil and paper. That would be like writing this chapter using a typewriter, rather than Microsoft Word! Not so bad if you get it all right the first time, but almost impossible to rework in the light of new ideas or mistakes.

But all you need for visual design is a good drawing package, right? One that will let you redraw as many times as you like? No, because a good modeling tool is more than just a drawing package — it actually understands the design you're creating. For example, if you draw a line between a shape representing a web application and a shape representing a Web service, the tool will be able to deduce the following:

- These shapes represent a web application and a Web service, respectively.
- The web application references the web service, and communicates with it in some way.

This understanding of your design enables a modeling tool to add value to the Software Development Lifecycle (SDLC) by automating the validation of your design, the generation of code, the deployment of applications, and the production of documentation. To put it another way, the modeling tool must understand your design in order to make model-driven development and Software Factories a reality.

## Meta-models

What distinguishes a good visual design (or modeling) tool from a simple drawing package? What is it that makes the tool understand your design? The meta-model.

Rational Rose has a true UML meta-model, accessible programmatically through the Rose Extensibility Interface (REI). Visio for Enterprise Architects ostensibly represents UML elements generically as pictorial shapes with no true meta-model, but it is possible to dig a little deeper to discover a meta-model of sorts encoded within those shapes.

> *Tony Loton's article "Build a Better Design Tool with Visio Automation" at* `www.asptoday.com/Content.aspx?id=2051` *demonstrates how to extract information from the Visio UML meta-model using VBA macros and .NET programs.*

IBM-Rational XDE has a meta-model partially represented in the code itself, as this is continually synchronized with the model, and also in an internal form accessible programmatically in a limited way through exposed COM interfaces.

Externally, modeling tools represent their meta-models using a proprietary file format or an industry standard format such as XML Metadata Interchange (XMI).

## SDM and the Team System meta-model

Because Class Designer synchronizes perfectly with classes in source code, on a one-for-one basis, the meta-model for this designer is in effect the programming language itself: Visual C#, Visual Basic, and Visual J# class diagram files carry very little additional metadata, a side-effect of which is that concepts which cannot be represented in code cannot be represented on diagrams. Therefore, whereas in Rational Rose class diagrams there is a diagrammatic distinction between associations and compositions, in Class Designer there is not.

With the Distributed System Designers, the situation is slightly different. They encode design information in XML conforming to the System Definition Model (SDM) schema. Try opening any of the application, logical datacenter, system, or deployment diagram files — with extensions `.ad`, `.ldd`, or `.sd` — using Windows NotePad, and you will see the following tag:

```
<SystemDefinitionModel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" Name="LogicalDataCenter2"
Version="1.0.0.0" Culture="en-US"
xmlns="http://schemas.microsoft.com/SystemDefinitionModel/2003/10">
```

As you progress through the software development life cycle, you will see that some of this information finds its way into separate files with an .sdm extension, again conforming to the same schema.

The good news is that you don't need to understand the SDM schema in order to use the modeling tools. For now, you just need to be aware that such a meta-model is important, and be mindful of the fact that everything you draw is described under-the-covers in the SDM format.

You can refer to Chapter 7 for more detailed information about the SDM.

# What about UML?

As a professional software designer, chances are good that you have at least a passing acquaintance with UML. If so, you might want to know the following: How can you capitalize on your investment in UML skills when adopting the new tools? How can you capitalize on your investment in UML artifacts? What if you really want a fully integrated UML capability?

We'll attempt to answer those questions, but first a few words about Microsoft's position regarding UML.

According to the published modeling strategy, Microsoft is not against UML as such. They see UML as a valuable notation for sketching out ideas early in the software life cycle, a task for which you can continue to use Visio in the retained role that we discuss shortly. They also support third-party vendors, some of whom are building UML 2.0 tools for Visual Studio 2005. In addition, where it makes sense to do so, the new set of visual designers will use UML-like notations to ease the transition.

However, Microsoft believes that it can better bridge the gap between design and development (modeling and coding) by providing a set of visual designers — underpinned by DSLs — that enable you to work at a high level of abstraction while providing sufficiently high fidelity for the specific domain being modeled. In this respect, using stereotypes and other mechanisms to extend UML 2.0 would have been overly complex and still too generic.

## *Capitalizing on your investment in UML skills*

A direct comparison between UML and the new visual designers is not valid, not to mention unfair. Microsoft has deliberately eschewed UML in favor of domain-specific languages that are highly tuned for specific tasks, and it's early days for this new vision, so we cannot expect a full end-to-end visual modeling capability in Visual Studio 2005.

However, just because a blow-by-blow comparison is not possible, you can still find similarities that offer opportunities to take advantage of your UML skills.

First, Application Designer enables you to model applications that communicate through endpoints in much the same way as you would use the UML component diagram to model components that communicate through interfaces. Therefore, if you're used to component diagrams, you'll take to application diagrams. In fact, we translated a UML component diagram into an equivalent VSTS application diagram early in this chapter.

Second, Logical Datacenter Design, in conjunction with Deployment Designer, enables you to model a logical infrastructure, and to show which applications will deploy to which servers, in much the same way as you would with a UML deployment diagram. If you're used to deployment diagrams, you'll take to logical datacenter diagrams.

When it comes to Class Designer, the similarities with UML class diagrams are apparent even from the first glance. The rendering of classes (as boxes with compartments) and interfaces (as lollipops) is much the same, as is some of the notation, such as the idea of "associations." In both cases, the purpose is exactly the same: to provide a one-for-one graphical representation of the underlying types in code. You just need to be aware of some terminology differences, such as "operations" and "attributes" in UML ("methods" and "fields" in Class Designer) and "generalization" in UML ("inheritance" in Class Designer).

We hope that those brief points have convinced you that all that UML training has not been a waste of time. You can take at least some of what you know and apply that knowledge when using the new tools.

You might have spotted that Visual Studio 2005 does not offer dynamic modeling capabilities that approximate UML's sequence and statechart diagrams. That situation might change over time as new visual designers are produced to support DSLs specifically for dynamic modeling. In the meantime, you can continue to use Visio (see the following section) or use strategically placed comments on diagrams to convey limited dynamic information.

## Capitalizing on your investment in UML artifacts

If you've already designed .NET applications, chances are good that you have some existing models in Visio for Enterprise Architects or IBM-Rational XDE that you'd like to migrate into the Team Architect.

Any attempt to migrate your models as a whole will prove fruitless. Not only because of the lack of XMI exchange features, but also because a significant proportion of the model's contents simply cannot be represented in Visual Studio 2005 — specifically, the use cases and the dynamic diagrams. That need not be a problem if Visio is used in the retained role indicated below, as the best tool for recording analysis-level dynamic information that does not directly affect the generated code.

You can certainly import into Visual Studio 2005 the static information from your existing models' class diagrams, either by generating code from Visio into Visual Studio or by reverse engineering existing deployed applications — including Web services — into Team Architect representations. Thanks to code synchronization, you should be able to quickly build a class diagram and/or application diagram that reflects the Visio-generated or reverse-engineered code base.

## Achieving a fully integrated UML capability

This chapter has offered some good reasons why you should not want this at all, and Microsoft can offer many more. However, you may still find yourself wanting to adopt Visual Studio 2005 Team System but unable to live without a UML capability.

One option is to wait for a third-party vendor to provide a UML add-in. Borland has announced an intention to develop UML 2.0 capabilities for the Visual Studio Team System.

The other option is to use Microsoft's DSL tools to grow your own UML designers. We don't recommend that unless you work for an organization that can absorb the development costs of such an initiative, or unless you think you can sell the end product! Having said that, growing your own design notations is considerably easier with the DSL tools than it ever used to be by other means. To prove it, we had a stab at devising a DSL to mirror the UML's activity diagram notation. The result is the Activity Designer shown in Figure 1-9.

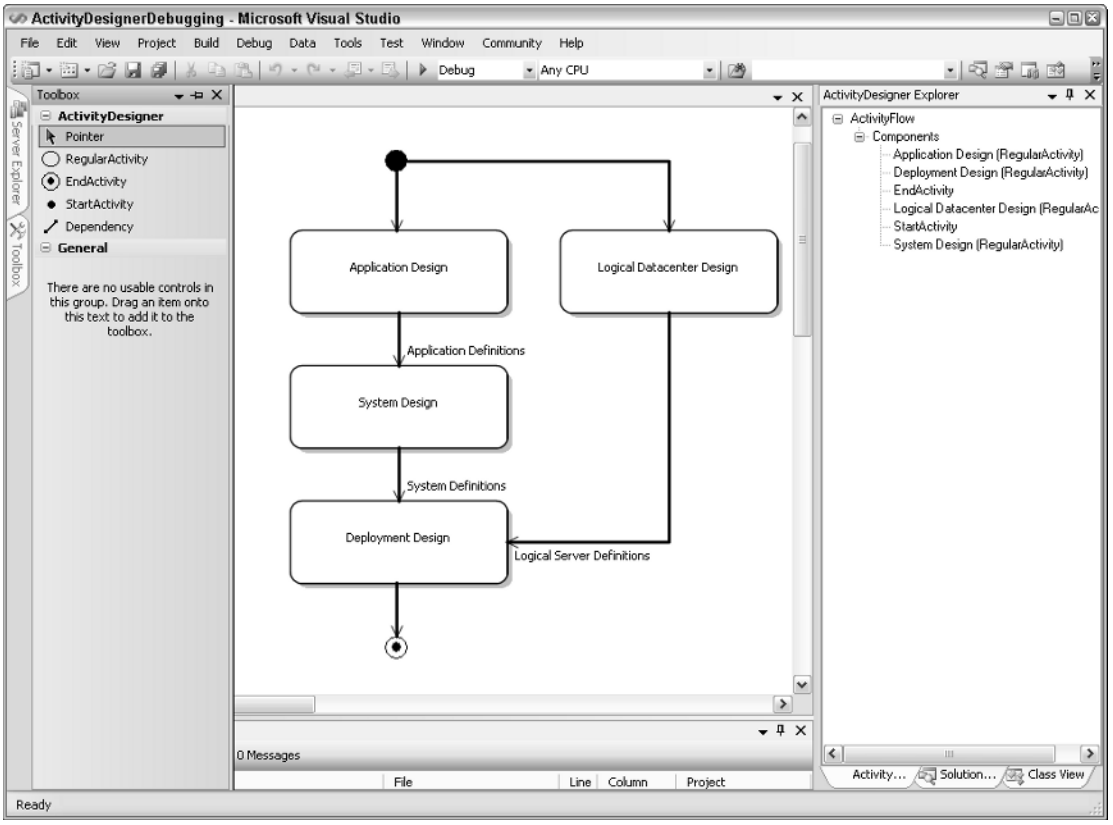


Figure 1-9

It turned out not to be production quality and it supported only a very limited subset of the notation, but when we tell you that it took only days — not weeks or months — to achieve something approaching the diagram we presented in Figure 1-8, you'll appreciate the potential offered by the DSL tools, even for UML stalwarts.

We could have just as easily used the DSL tools to build simulations of the other UML diagram types, such as use case diagrams and collaboration diagrams. And if all that sounds like too much hard work, you'll be pleased to hear that Microsoft's DSL Tools team has provided you with a starting point by bundling sample DSLs for UML use case diagrams, activity diagrams, and class diagrams with the DSL toolkit available at `http://lab.msdn.microsoft.com/teamsystem/workshop/dsltools/default.aspx`.

## A retained role for Visio for Enterprise Architects

Previously, Microsoft bundled Visio for Enterprise Architects with Visual Studio .NET, as the preferred tool for modeling .NET applications. Microsoft will continue to bundle Visio for Enterprise Architects as a complementary tool providing a broad modeling capability, most useful in the early analysis stages, and compliant with UML 1.3.

Here's our advice on how you should continue to use Visio:

What we're *not* advocating is using Visio in combination with the new visual designers for detailed design, implementation, and deployment activities within Visual Studio. The Team Edition for Software Architects is clearly stronger here with its IDE integration, code synchronization, and bias toward creating deployable designs.

What we are advocating is using Visio for the up-front analysis and design activities that may well be undertaken by a business or systems analyst already familiar with working with Visio, rather than a designer/developer familiar with working with Visual Studio. In a nutshell, we propose using the right tool for each job, with little or no overlap, as follows:

- Use Visio for the initial analysis activities, which in any case may be performed by people — business analysts or solution designers — more used to working in Visio than in Visual Studio. We're talking about use cases and scenario realizations (e.g., as sequence diagrams) that are in any case not supported by the Team Architect.
- Use Visio to generate a one-time-only domain model as code. Because the Team Architect keeps code and model in perfect sync, any Visio-generated classes will be available immediately to your Visual Studio model.
- Use the Visual Studio 2005 Application Designer to sketch out the overall shape of the system in terms of interconnected applications and services, analogous to a UML component diagram.
- Use the Visual Studio 2005 Class Designer to evolve the Visio-generated domain model (if you have one) into a proper technical design model incorporating the .NET framework classes that are not supported well in Visio.
- Develop the code itself, with Visual Studio 2005 automatically ensuring synchronization between the code and the model, analogous to UML round-trip engineering.
- Use the Visual Studio Logical Datacenter Designer and Deployment Designer to specify the mappings and constraints for the deployed system, analogous to a UML deployment diagram.

Finally, if you need to do any limited dynamic modeling in support of the detailed designs, do this using strategically placed comments to convey the necessary information without reverting back to Visio.

> **The UML capabilities of Visio are fully documented in *Professional UML with Visual Studio.NET: Unmasking Visio for Enterprise Architects* (Wrox, 2003). This book is based on Visio for Enterprise Architects 2003, but for the retained role we suggest it should be equally applicable to the newer Visio for Enterprise Architects 2005.**

# Summary

In this chapter we first set the scene by establishing the case for doing design, specifically visual design, at all. Because Microsoft's new vision for how visual design should work is somewhat different from the old-world view based around UML, we highlighted the three main pillars that support that vision — namely, model-driven development (MDD), domain-specific languages (DSL), and software factories.

Because the underlying technologies have also evolved over time, thereby influencing the kinds of design you'll produce, we also traced that evolution of concepts from the original object-oriented paradigm through components and distributed-components to the service-oriented architectures that represent the current wisdom. We did that for a very good reason — as a lead-in to the Application Designer, which lends itself to service-based application design.

Application Designer is the first in the "Whitehorse" set of Distributed System Designers, which also includes Logical Datacenter Designer, System Designer, and Deployment Designer. We introduced each one and showed the important relationships between them. We also introduced Class Designer, which is not strictly part of the same set but which is complementary — and similar — in use.

Because all of the designers apart from Class Designer are tied together by a meta-model in the form of the System Definition Model (SDM), we introduced the term SDM and invited you to refer to Chapter 7 for further information.

We'd be failing you if we had not paid some attention to the hitherto industry-standard notation for visual modeling (UML). Our aim was not to compare the UML approach with Microsoft's vision, but to help you make the transition by suggesting which skills and artifacts might be transferable. Because Visual Studio doesn't yet provide a complete end-to-end modeling solution, you learned how you might continue to use Visio for Enterprise Architects in the early phases of the software development life cycle.

Having introduced the new visual designers in this chapter, we now devote Chapters 2 through 5 to covering them in detail.