1
Domain-Specific Development

Introduction

This book describes the Microsoft Domain-Specific Language Tools (the DSL Tools). The DSL Tools are part of the Visual Studio SDK, and may be downloaded from http://msdn.microsoft.com/vstudio/DSLTools/. The DSL Tools extend Microsoft Visual Studio 2005 to support a powerful way of developing software called Domain-Specific Development.

Domain-Specific Development is based on the observation that many software development problems can more easily be solved by designing a special-purpose language. As a small example, think about the problem of finding every occurrence of a particular pattern of characters in a file, and doing something with each occurrence that you find. The special-purpose textual language of *regular expressions* is specifically designed to do this job. For example, using the .NET class `System.Text.RegularExpressions.Regex`, the regular expression `(?<user>[^@]+)@(?<host>.+)` applied to a string of characters will find email addresses in it, and for each address found, assign the substring immediately before the `@` sign to the `user` variable, and the substring immediately after the `@` sign to the `host` variable. Without the regular expression language, a developer would have to write a special program to recognize the patterns and assign the correct values to the appropriate variables. This is a significantly more error-prone and heavyweight task.

Domain-Specific Development applies this same approach to a wide variety of problems, especially those that involve managing the complexity

of modern distributed systems such as those that can be developed on the .NET platform. Instead of just using general-purpose programming languages to solve these problems one at a time, the practitioner of Domain-Specific Development creates and implements special languages, each of which efficiently solves a whole class of similar problems.

Domain-Specific Languages can be textual or graphical. Graphical languages have significant advantages over textual languages for many problems, because they allow the solution to be visualized very directly as diagrams. The DSL Tools make it easy to implement graphical DSLs, and they enable Domain-Specific Development to be applied to a wide range of problems.

Domain-Specific Development

Domain-Specific Development is a way of solving problems that you can apply when a particular problem occurs over and over again. Each occurrence of the problem has a lot of aspects that are the same, and these parts can be solved once and for all (see Figure 1-1). The aspects of the problem that are different each time can be represented by a special language. Each particular occurrence of the problem can be solved by creating a model or expression in the special language and plugging this model into the fixed part of the solution.

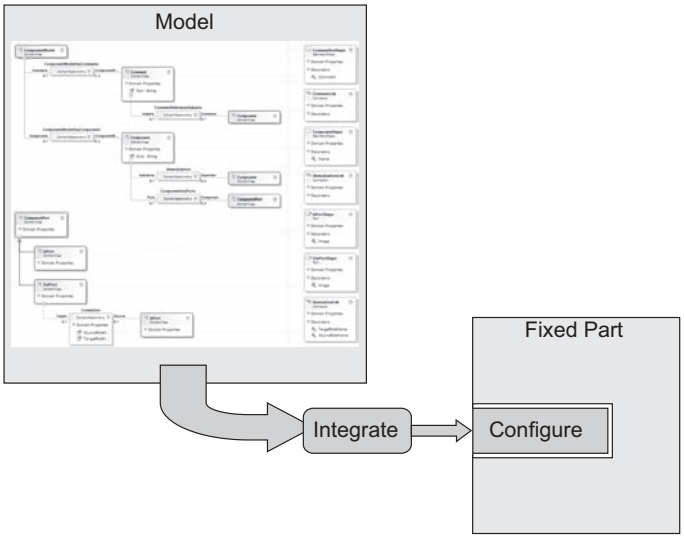

FIGURE 1-1: Domain-Specific Development

The fixed part of the solution is written using traditional design, coding, and testing techniques. Depending on the size and shape of the problem, this fixed part of the solution might be called a framework, a platform, an interpreter, or an Application Programming Interface (API). The fixed part captures the architectural patterns that make up the domain and exposes extension points that enable it to be used in a variety of solutions. What makes the approach applicable is the fact that you create the variable part of the solution by using a special-purpose language—a DSL.

As we observed in the introduction, the DSL might be textual or graphical. As the technology for domain-specific development matures, we expect to see tools that support the development and integration of both textual and graphical DSLs. People have a range of feelings about which kind of language they prefer. Many people, for example, prefer textual languages for input, because they can type fast, but graphical languages for output, because it is easier to see the "big picture" in a diagram. Textual expressions make it much easier to compute differences and merges, whereas graphical expressions make it much easier to see relationships. This chapter discusses both kinds, but the first version of DSL Tools and hence the remaining chapters of the book focus solely on graphical languages.

To create a working solution to the problem being addressed, the fixed part of the solution must be integrated with the variable part expressed by the model. There are two common approaches to this integration. First, there is an interpretative approach, where the fixed part contains an interpreter for the DSL used to express the variable part. Such an approach can be flexible, but it may have disadvantages of poor performance and difficulty in debugging. Second, the particular expression or diagram may be fully converted into code that can be compiled together with the remainder of the solution—a code-generation approach. This is a more complex conversion procedure, but it provides advantages in extensibility, performance, and debugging capability.

Graphical DSLs are not just diagrams. If you wanted just to create diagrams, you could happily use popular drawing programs such as Microsoft Visio to achieve a first-class result. Instead, you are actually creating models that conceptually represent the system you are building, together with diagrammatic representations of their contents. A given model can be represented simultaneously by more than one diagram, with each diagram representing a particular aspect of the model, as shown in Figure 1-2.

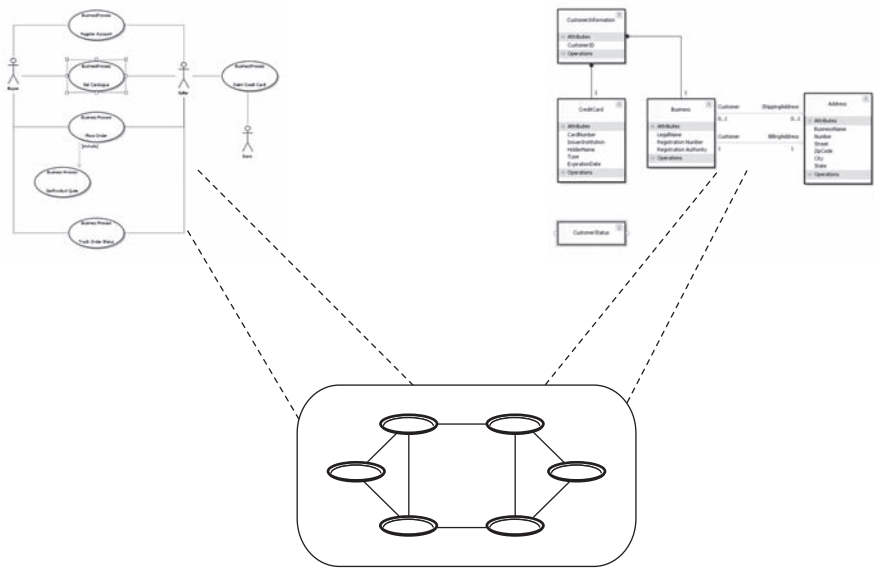

FIGURE 1-2: Two diagrams and one model

Examples

Let's first have a look at a couple of examples where the DSL Tools have been applied in practice. The first example comes from an Independent Software Vendor (ISV) called Himalia. Himalia has created a set of DSLs for implementing complex graphical user interfaces without doing any coding. The Himalia Navigation Model, shown in Figure 1-3, defines the navigation through the user interface.

Use Cases, regarded as heavyweight flows of control consisting of activities and transitions, are explicitly defined in a state machine view in order to address their complexity. Use Case states and transitions are related to Navigation Model elements and actions, respectively. The Use Case Model is shown in Figure 1-4.

The User Profile Model shown in Figure 1-5 defines user states that affect the behavior of the user interface.

The complete Himalia system integrates these models with others into Visual Studio 2005 to implement complete user interfaces based on Microsoft technology, including Windows Presentation Foundation (WPF).

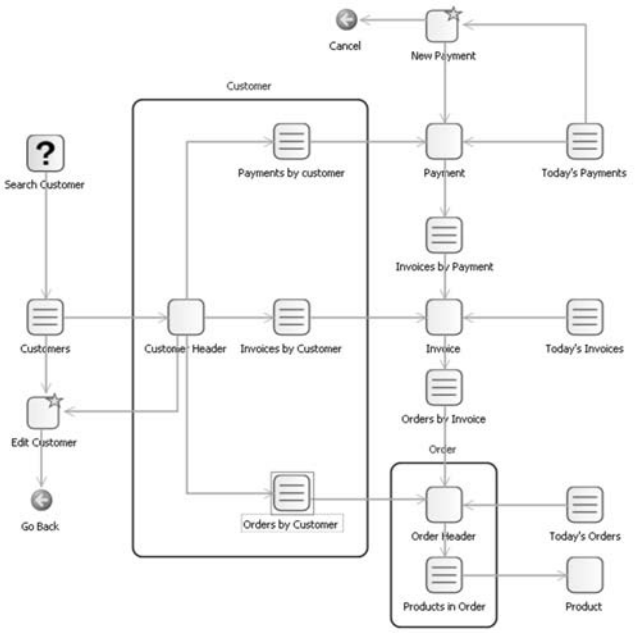

FIGURE 1-3: Himalia Navigation Model

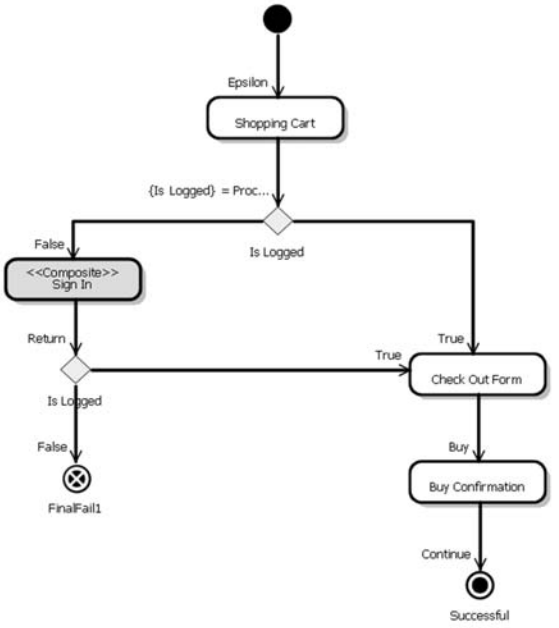

FIGURE 1-4: Himalia Use Case Model

The second example is a Systems Integrator (SI) called Ordina that is based in the Netherlands. Ordina has built a complete model-driven software

factory within its Microsoft Development Center, called the SMART-Microsoft Software Factory. This factory uses four connected DSLs. To enable these DSLs to collaborate, Ordina has created a cross-referencing scheme that allows elements in one DSL to refer to elements in another DSL.

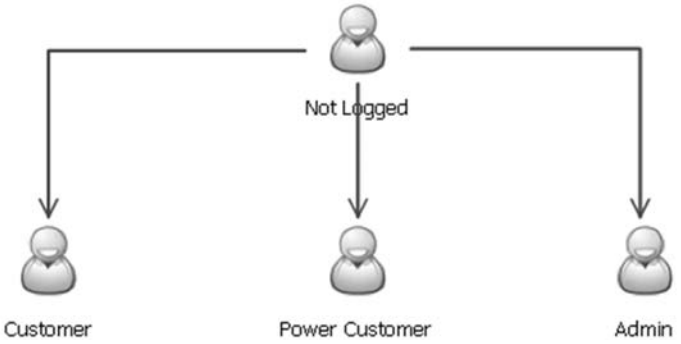

FIGURE 1-5: Himalia User Profile Model

The Web Scenario DSL is used to model web pages and user actions, and to generate ASP.NET web pages. An example is shown in Figure 1-6.

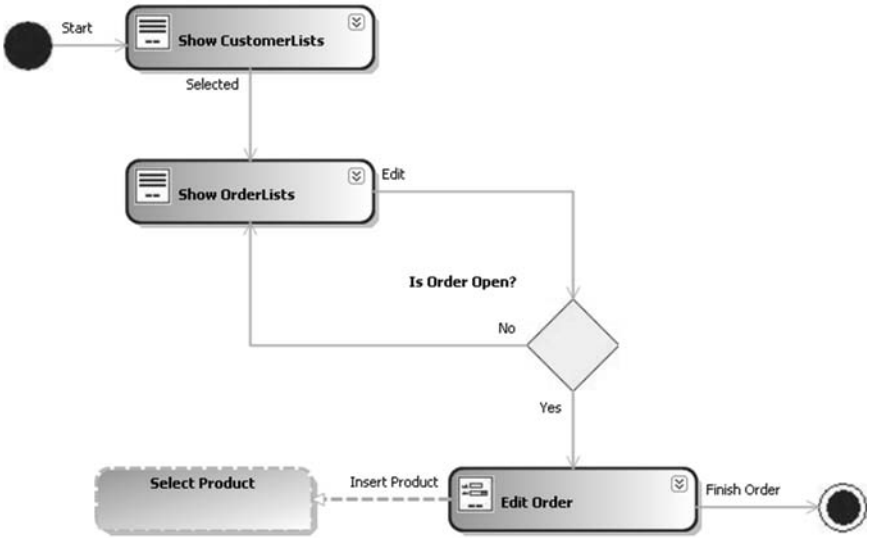

FIGURE 1-6: Ordina Web Scenario DSL

The Data Contract DSL is used to define the data objects that are transferred between the different layers in the architecture. An example is shown in Figure 1-7, which illustrates several different kinds of data objects.

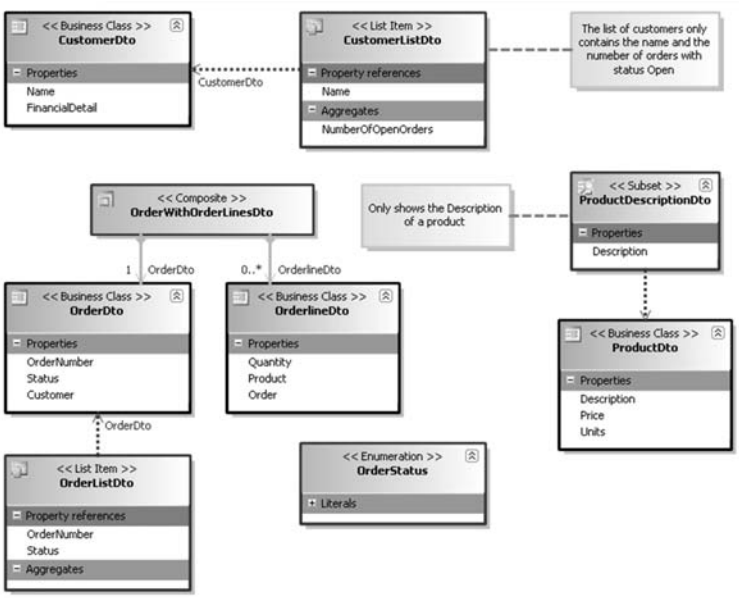

FIGURE 1-7: Ordina Data Contract DSL

The third DSL in the Ordina factory is the Service Model shown in Figure 1-8, which is used to generate service interfaces and skeletons of the business processes that implement the services.

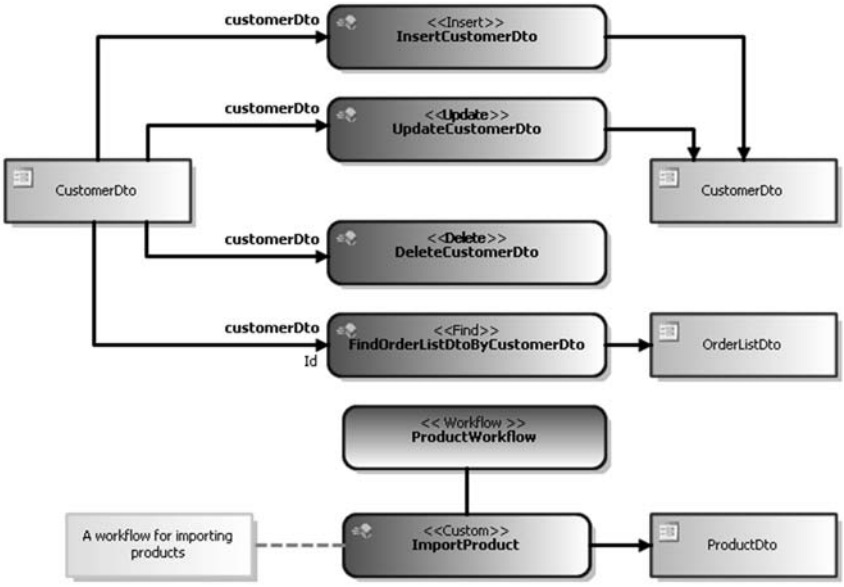

FIGURE 1-8: Ordina Service DSL

The final DSL in the Ordina factory is the Business Class Model that is used to generate code for the Business Class and Data layers. This model is shown in Figure 1-9.

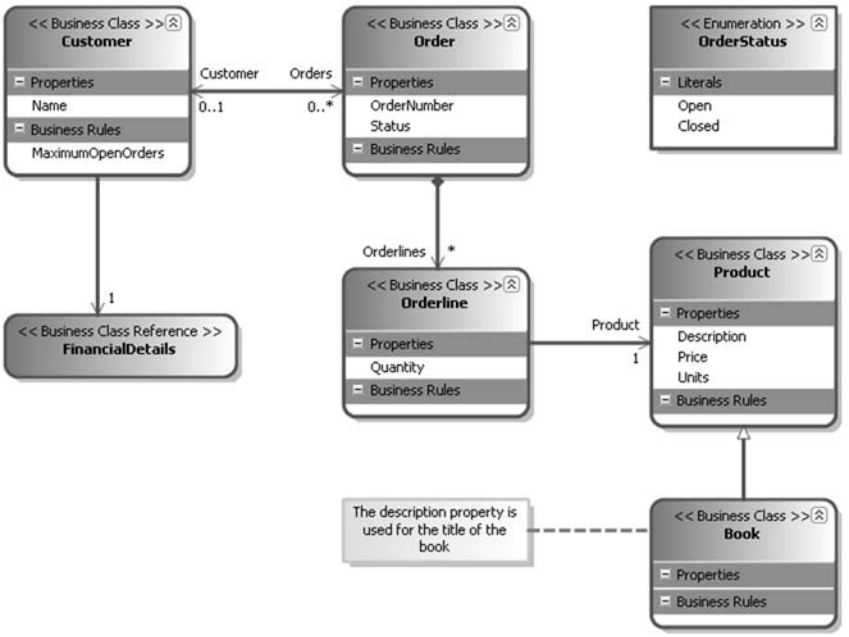

FIGURE 1-9: Ordina Business Class DSL

These two examples from Himalia and Ordina are for "horizontal" DSLs, where the intended customer for the resulting software does not belong to any particular industry. Here are some other more "vertical" examples of where domain-specific development might be applied.

Software Defined Circuitry

Many electronic products have circuitry that is programmed using software. For example, FPGAs (Field Programmable Gate Arrays) are programmable chips used in areas such as software defined radio, digital signal processing, medical imaging and speech recognition. Programming such chips directly in their Hardware Description Language (HDL) is a very low-level and painstaking task. A Domain-Specific Development approach can be used to raise the level of abstraction until it represents

much more directly the domain being implemented; for example, a DSL approach to software defined radio is discussed in the paper by Bruce Trask of PrismTech at www.mil-embedded.com/articles/authors/trask/.

Embedded Systems

Many real-time embedded systems can be conceptualized as a set of communicating finite state machines. Separating the design of these systems into explicit state machines, plus a generic platform for executing state machines, can greatly simplify thinking about such systems. In this case, the DSL is the language for expressing state machines consisting of states and the transitions between them, while the execution platform is most likely built using custom code.

Device Interfaces

Many modern devices, such as mobile phones, HiFi equipment, and so on, have complex user interfaces. These interfaces are typically organized via rules that make the interface predictable, such as a rule that pressing a cancel button always takes you back to a known state, or inputting text always follows the same set of predictive rules. A DSL can be created for designing such systems, where the graphical appearance of the language corresponds accurately to the appearance of the actual interface being designed, and the interaction rules of the interface are captured in the structure of the language. Good examples of this approach can be found at the Domain-Specific Modeling Forum website at www.dsmforum.org.

Software Development Process Customization

The example that is used throughout this book to illustrate the DSL Tools shows how to use DSLs to define aspects of a software development process, such as the processing of bugs and issues, and how to use the models to configure the tools used to enact the process.

All of these examples and many others share the same approach: (1) identifying aspects of the problem that are fixed for all occurrences and capturing those aspects in a common framework or platform, and (2) identifying the other aspects that vary between occurrences and designing a Domain-Specific Language whose expressions or models will specify a solution to the problem.

Benefits

Now that we've looked at some examples, we can see the benefits of Domain-Specific Development.

- A DSL gives the ability to work in terms of the problem space, with less scope for making the errors that come from representing it in a general-purpose language.
- Working in terms of the problem space can make the models more accessible to people not familiar with the implementation technology, including business people.
- Models expressed using DSLs can be validated at the level of abstraction of the problem space, which means that errors in understanding or representation can be picked up much earlier in the development lifecycle.
- Models can be used to simulate a solution directly, providing immediate feedback on the model's suitability.
- Models can be used to configure an implementation consisting of multiple technologies of different types, which can reduce the skill and effort required to implement a solution using these technologies.
- Models can also be used to generate other models, and to configure other systems, networks, and products, perhaps in combination with other enabling technologies such as wizards.
- A domain-specific language provides a domain-specific API for manipulating its models, thus improving developer productivity.
- The artifacts generated from a DSL need not all be technological implementation artifacts; a suitable model can be used to generate build scripts, purchase orders, documentation, bills of materials, plans, or skeletons of legal contracts.
- Once important business knowledge is captured in a model, it becomes considerably easier to migrate a solution from one technology to another, or between versions of the same technology. This can often be done simply by modest modifications to the generators or interpreter.

In combination, these factors can offer considerable increased agility. For example, in the software defined radio domain mentioned earlier, Bruce Trask reports that "users of the tool report a minimum of 500 percent increase in productivity."

Of course these benefits are not free. To get them, you must invest in designing and building a DSL and integrating it into your overall solution. This will involve the cost of development—which is considerably reduced using DSL Tools. But it will also include costs for testing, deployment, documentation, staff training, development process modifications, and so on. When setting out to implement a DSL you must balance these costs against the expected benefits. You'll get the benefits when the costs can be paid off from the benefits of applying the approach to lots of systems. Hence the approach is particularly attractive to Systems Integrators, who often have to carry out many similar software development engagements for one customer after another. For a small company that does not specialize in particular business areas, it may be worth investing in DSLs that describe technological domains, such as web services and databases; for a larger company that is vertically organized into industry specializations, it may also be worth investing in DSLs that describe corresponding business domains.

Languages

At this point, we offer a definition of Domain-Specific Language:

> A Domain-Specific Language is a custom language that targets a small problem domain, which it describes and validates in terms native to the domain.

Most computer languages are textual, with their statements and expressions consisting of sequences of characters in a standard character set. Graphical languages have become increasingly popular in recent years, particularly with the emergence of the Unified Modeling Language (UML) as a popular set of standard notational conventions for depicting elements in an object-oriented software system.

When computer experts talk about languages, they usually mean general-purpose textual programming languages such as Visual Basic, C#, or Java. In Domain-Specific Development, our interpretation of the word *language* is widened considerably—it includes graphical languages such as UML, flowcharts, entity-relationship diagrams, state diagrams, Venn diagrams, and so on. We also include other textual languages such as XML and domain-specific varieties like SQL and regular expressions. We even think of tabular and form-based formats such as spreadsheets or the Windows Forms Designer as being languages. Special languages also exist for domains such as music notation and direct-manipulation interfaces. With the power available in modern computers, there is absolutely no need to be restricted to simple linear textual notations to convey our intentions to the computer; we want to exploit the power of the computer to provide means to express the author's intent as directly as possible, thus increasing the efficiency of our development. This includes interactive aspects such as dragging and other gestures, context menus, toolbars, and so on.

There are two main forces at work driving the evolution of languages. The first of these is the progressive lifting of the level of abstraction at which we express our intentions about what we want the computer to do. Originally, programmers had to express their algorithms and data structures in terms directly accessible to the computer hardware, which was efficient for the hardware but very tedious and error-prone for the programmer. Subsequent developments such as symbolic assemblers, filing systems, third- and fourth-generation languages, databases, class libraries, and model-driven development have moved the languages in which developers express their intentions further from the computer hardware and closer to the problems they are trying to solve.

The second force driving language evolution is the increasing variety of available digital media. Originally, computers were used purely to compute with numbers, then also with symbols and texts, and then with bitmaps and images. The evolution of computing has reached a point where the limitation on how we express our intentions is no longer the physical capabilities of the computer itself but the limits of our understanding of how to construct and manipulate computer languages. In Domain-Specific Development, instead of building on a general-purpose language in order to solve a problem, we use a language that is *itself* designed to suit the problem being solved.

Related Work

Domain-Specific Development is not new. In 1976, David Parnas introduced the concept of families of programs in his paper "On the Design and Development of Program Families" and talked about the possibility of using a program generator to create the family members. In 1986, Jon Bentley in his column in the journal *Communications of the ACM* pointed out that much of what we do as programmers is the invention of "little languages" that solve particular problems. Later, in 1994, the popular and seminal book *Design Patterns: Elements of Reusable Object-Oriented Software*, by Gamma, Helm, Johnson, and Vlissides (also known as the "Gang of Four" book), introduced the *Interpreter* pattern. According to the authors, the intent of this pattern is: "Given a language, define a representation of its grammar along with an interpreter that uses the representation to interpret sentences in the language." But it is only relatively recently that Domain-Specific Development has begun to gain widespread acceptance in the IT industry.

Domain-Specific Development is closely related to many emerging initiatives from other authors and organizations, of which the following is a partial list.

Model-Driven Development

Many vendors of software development tools are offering Model-Driven Development tools, which allow users to build a model of their problem, often using a graphical language such as the Unified Modeling Language (UML). From these models, a code generator or model compiler is used to generate some or all of the code for the resulting application. The Object Management Group has a branded initiative under this heading called Model Driven Architecture (MDA). We'll talk more about model-driven development and MDA later in this chapter.

Language-Oriented Programming

Sergey Dimitriev, co-founder and CEO of JetBrains, uses the term "Language Oriented Programming" to describe the approach of creating a domain-specific language to solve a programming problem in his article "Language-Oriented Programming: The Next Programming Paradigm" at www.onboard.jetbrains.com/is1/articles/04/10/lop/.

Language Workbenches

Martin Fowler, popular industry author and speaker, also refers to Language-Oriented Programming and uses the term "Language Workbench" to refer to the kind of tools required to support Language-Oriented Programming and Domain-Specific Development in his article "Language Workbenches: The Killer App for Domain-Specific Languages?" at http://martinfowler.com/articles/languageWorkbench.html.

Domain-Specific Modeling

The Domain-Specific Modeling Forum (www.dsmforum.org) is a body that promotes the idea of specifying domain-specific languages and generating solutions from them. Their site contains several interesting and compelling case studies.

Generative Programming

The book *Generative Programming: Methods, Tools, and Applications,* by Krzysztof Czarnecki and Ulrich W. Eisenecker, discusses how to automate the generation of applications, with a particular focus on domain engineering and feature modeling, and presents a detailed discussion of several different techniques for program generation. There is a regular conference called Generative Programming and Component Engineering (GPCE) dedicated to this topic.

Intentional Software

Intentional Software (www.intentionalsoftware.com) aims to develop an environment in which all programming is domain-specific. Its Domain Workbench technology represents programs and models as data, and provides multiple ways to render and interact with them using domain-specific textual and graphical syntax.

Software Factories

Software Factories are described in the book *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, by Jack Greenfield and Keith Short, with Steve Cook and Stuart Kent. Software Factories are a strategic initiative from Microsoft that proposes to use a combination of passive content such as patterns, models, DSLs, assemblies,

and help files, with dynamic content such a customized tools, tailored processes, templates, wizards, and tests, all integrated into Visual Studio for producing a particular type of solution. DSL Tools form an important part of this initiative.

Textual DSLs

Before talking about graphical DSLs, let's look briefly at textual DSLs. We'll see how Domain-Specific Development involves a particular way of thinking about a problem, and we'll look at how to implement this approach using textual languages.

Imagine that we are designing a graphical modeling tool and have the problem of defining a set of shapes that will be displayed on a screen to represent the various concepts that can be depicted by the tool. One way we might do this would be to invent a new textual language for defining the various shapes. A fragment of this language might look like this:

```
Define AnnotationShape Rectangle
      Width=1.5
      Height=0.3
      FillColor=khaki
      OutlineColor=brown
      Decorator Comment
            Position="Center"
      End Comment
End AnnotationShape
```

In order to process this language, a program must be written to parse and interpret this text. As a programming exercise from scratch, this is a big job. But a parser-generator might be used, which itself takes as input a description of the grammar of the new language, such as the following, based on BNF (the Backus Naur Form, originally developed for defining the Algol language):

```
Definitions ::= Definition*
      Definition ::= Define Id Shape
      Width Eq Number
      Height Eq Number
```

```
      FillColor Eq Color
      OutlineColor Eq Color
      Decorator*
End Id

Shape ::= Rectangle | RoundedRectangle | Ellipse

Eq ::= "="

Decorator ::= Decorator Id
      Position Eq Position
End Id

Position ::= Center|
             TopLeft |
             TopRight |
             BottomLeft |
             BottomRight
```

The definitions for Id, Number, and Color are not included here; it's assumed that they are built into the grammar-defining language.

We need an algorithm to convert this BNF into a parser for the language it describes. We'd either use an existing parser-generator such as Yacc, Bison, Antlr, or Happy, or an expert might write one by hand in a normal third-generation programming language such as C# or Java.

Notice that the BNF is itself a DSL. We might "bootstrap" the BNF language by describing its grammar in itself, causing it to generate a parser for itself. Perhaps the hand-written parser will be quite simple, and the generated parser would handle a more complicated version of BNF. This pattern of using languages to describe languages, and bootstrapping languages using themselves, is very common when defining domain-specific languages.

Implementing a textual DSL by implementing its grammar like this can be a difficult and error-prone task, requiring significant expertise in language design and the use of a parser-generator. Implementing a parser-generator is definitely an expert task, because a grammar might be ambiguous or inconsistent, or might require a long look-ahead to decide what to do. Furthermore, there is more to implementing a language than just implementing a parser. We'd really like an editor for the language that gives the kinds of facilities we expect from a programming language editor in a modern development

environment, like text colorization, real-time syntax checking, and auto-completion. If you include these facilities, the task of implementing a textual language can get very large. Happily, there are alternative strategies for implementing a textual DSL that don't involve implementing a new grammar.

The first strategy is to use the facilities of a host language to emulate the capabilities of a domain-specific language. For example, the following C# code has the effect of defining the same shape as the previous example:

```
Shape AnnotationShape = new Shape(ShapeKind.Rectangle,
                                  1.5,
                                  0.3,
                                  Color.Khaki,
                                  Color.Brown);
Decorator Comment = new Decorator(Position.Center);
AnnotationShape.AddDecorator(Comment);
```

This kind of code is often called *configuration code*, because it uses previously defined classes and structures to create a specific configuration of objects and data for the problem that you want to solve. In effect, the definitions of these classes and structures are creating an *embedded DSL*, and the configuration code is using that DSL. The capabilities of modern languages to define abstractions such as classes, structures, enumerations, and even configurable syntax make them more amenable to this approach than earlier languages that lacked these facilities.

The second strategy is to use XML—Extensible Markup Language. There are many ways in which the definition can be expressed using XML. Here's a possible approach.

```
<?xml version="1.0" encoding="utf-8" ?>
<Shapes>
  <Shape name="AnnotationShape">
    <Kind>Rectangle</Kind>
    <Width>1.5</Width>
    <FillColor>Khaki</FillColor>
    <OutlineColor>Brown</OutlineColor>
    <Decorator name="Comment">
      <Position>Center</Position>
    </Decorator>
  </Shape>
</Shapes>
```

The syntax is obviously limited to what can be done using XML elements and attributes. Nevertheless, the tags make it obvious what each element is intended to represent, and the meaning of the document is quite clear. One great advantage of using XML for this kind of purpose is the widespread availability of tools and libraries for processing XML documents.

If we want to use standard XML tools for processing shape definitions, the experience will be much improved if we create a schema that allows us to define rules for how shape definitions are represented in XML documents. There are several technologies available for defining such rules for XML documents, including XML Schema from the World Wide Web Consortium (defined at www.w3.org/XML/Schema.html), RELAX NG from the OASIS consortium (defined at www.relaxng.org) and Schematron, which has been accepted as a standard by the International Organization for Standardization (ISO) and is defined at www.schematron.com. Schematron is supported in .NET: A version called Schematron.NET is downloadable from SourceForge, and it is possible to combine the facilities of XML Schema and Schematron. We'll use here the XML Schema approach, which is also supported by the .NET framework.

An XML Schema is an XML document written in a special form that defines a grammar for other XML documents. So, using an appropriate schema, we can specify exactly which XML documents are valid shape definition documents. Modern XML editors, such as the one in Visual Studio 2005, can use the XML schema to drive the editing experience, providing the user with real-time checking of document validity, colorization of language elements, auto-completion of tags, and tips about the document's meaning when you hover above the elements.

Here is one of many possible XML schemas for validating shape definition documents such as the one presented earlier. Writing such schemas is something of an art; you'll certainly observe that it is significantly more complicated than the BNF that we defined earlier, although it expresses roughly the same set of concepts.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema
  xmlns="http://schemas.microsoft.com/dsltools/ch01"
  attributeFormDefault="unqualified"
  elementFormDefault="qualified"
```

```
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://schemas.microsoft.com/dsltools/ch01">
  <xs:element name="Shapes">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="Shape">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Kind" type="kind" />
              <xs:element name="Width" type="xs:decimal" />
              <xs:element name="Height" type="xs:decimal" />
              <xs:element name="FillColor" type="xs:string" />
              <xs:element name="OutlineColor" type="xs:string" />
              <xs:element maxOccurs="unbounded" name="Decorator">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="Position" type="position" />
                  </xs:sequence>
                  <xs:attribute name="name" type="xs:string" use="required" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:simpleType name="position">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Center" />
      <xs:enumeration value="TopLeft" />
      <xs:enumeration value="TopRight" />
      <xs:enumeration value="BottomLeft" />
      <xs:enumeration value="BottomRight" />
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="kind">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Rectangle" />
      <xs:enumeration value="RoundedRectangle" />
      <xs:enumeration value="Ellipse" />
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

To summarize, in this section we have looked at three ways of defining a textual DSL: using a parser-generator, writing configuration code embedded in a host language, and using XML with a schema to help validate your documents and provide syntax coloring and autocompletion. A further option would be to define an equivalent to the DSL Tools that targeted textual languages.

Each of these approaches has its pros and cons, but they all share a common theme—investing some resources early in order to define a language that will make it easier to solve specific problems later. This is the basic pattern that also applies to graphical DSLs, as we shall see.

The DSL Tools themselves provide no facilities in version 1 for defining textual domain-specific languages. The Tools' authors have taken the view that XML provides a sufficiently good approach to start with, and so they have designed the DSL Tools to integrate XML-based textual DSLs with graphical DSLs.

Graphical DSLs

So far we have looked at some of the background behind Domain-Specific Development and discussed its benefits. We have also looked briefly at textual DSLs. Let's start our exploration into graphical DSLs by looking at an example that captures data for deploying and managing distributed applications.

Figure 1-10 shows a simple model built using a graphical DSL for designing logical data centers. This DSL is part of Visual Studio 2005 Team Architect. The elements of this language include *zones*, depicted by rectangular areas surrounded by dashed lines; *hosts*, depicted by rectangular areas surrounded by solid lines; *endpoints*, depicted by small shapes (squares, circles, and hexagons) placed on the edges of hosts; and *connections*, depicted by arrows between endpoints. This model corresponds exactly to an XML file that contains information according to the rules of the System Definition Model (SDM), which is used for configuring and managing data centers.

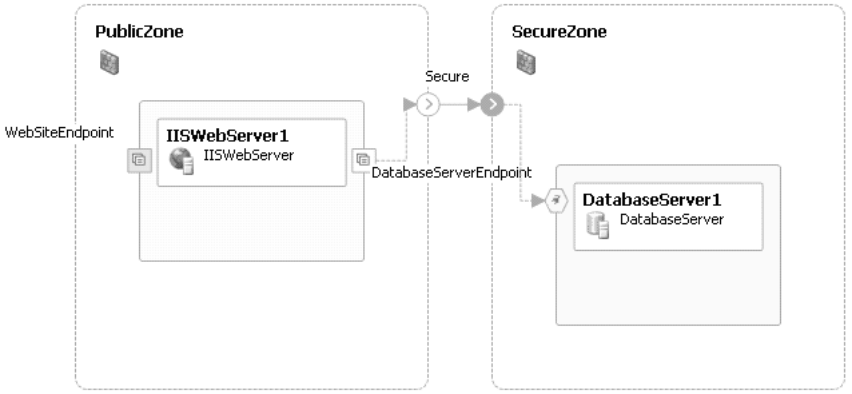

FIGURE 1-10: Data center design

System Definition Model

SDM was created as part of Microsoft's Dynamic Systems Initiative, which promises to deliver self-managing dynamic systems that will result in reduced operational costs and increased business agility. A later version of this model, called SML (Service Modeling Language), is being standardized by industry leaders, which should eventually enable distributed systems with components from multiple vendors to be managed using these models.

We can build up graphical DSLs like this one from a set of simple diagrammatic conventions such as the following. Many of these conventions are derived from UML, which we discuss in more depth later.

Conventions for Representing Structure

See Figure 1-11 for examples of structural conventions, including:

- Nested rectangle or rounded rectangles, to represent structural containment
- Rectangles with headers, sections, and compartments, to represent objects, classes, entities, devices, and so on
- Solid and dashed connectors with multiplicities, names, arrowheads, and other adornments, to represent relationships, associations, connections, and dependencies

- Connectors with large open triangular arrowheads, to represent generalization, inheritance, and derivation
- Ports on the edges of shapes, to represent connectable endpoints

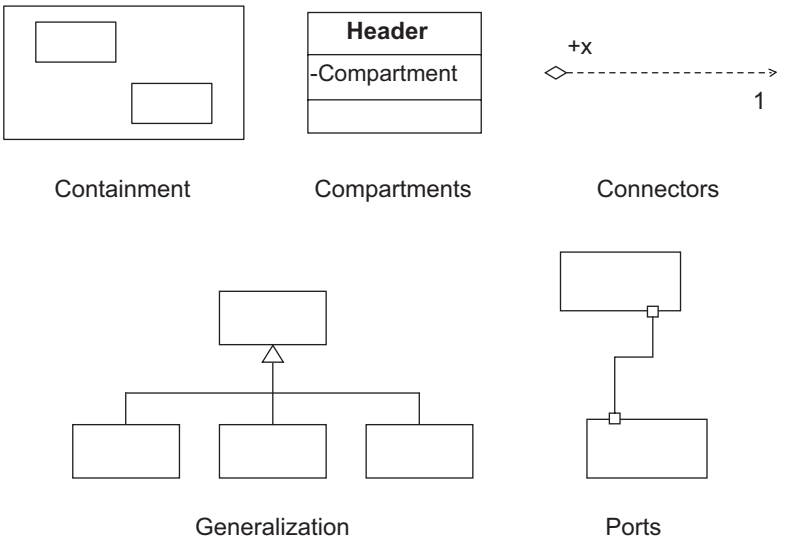

FIGURE 1-11: Structural conventions

Conventions for Representing Behavior

See Figure 1-12 for examples of behavioral conventions, including:

- Lifelines and arrows, to represent sequences of messages or invocations with a temporal axis
- Rounded rectangles, arrows, swimlanes, diamonds, transition bars, and so on, to represent activities and flows
- Nested ovals and arrows, to represent states and transitions
- Ovals and stick people, to represent use cases and actors

Using the DSL Tools, it is possible to build your own graphical language that combines conventions like these in a way that matches your particular problem (although version 1 of the Tools does not fully support all of the conventions listed). You can map them onto the concepts of your own domain and construct a customized graphical modeling language that solves your own problem. We saw an example in the data center design language shown in Figure 1-10, and we'll see many other examples as we proceed.

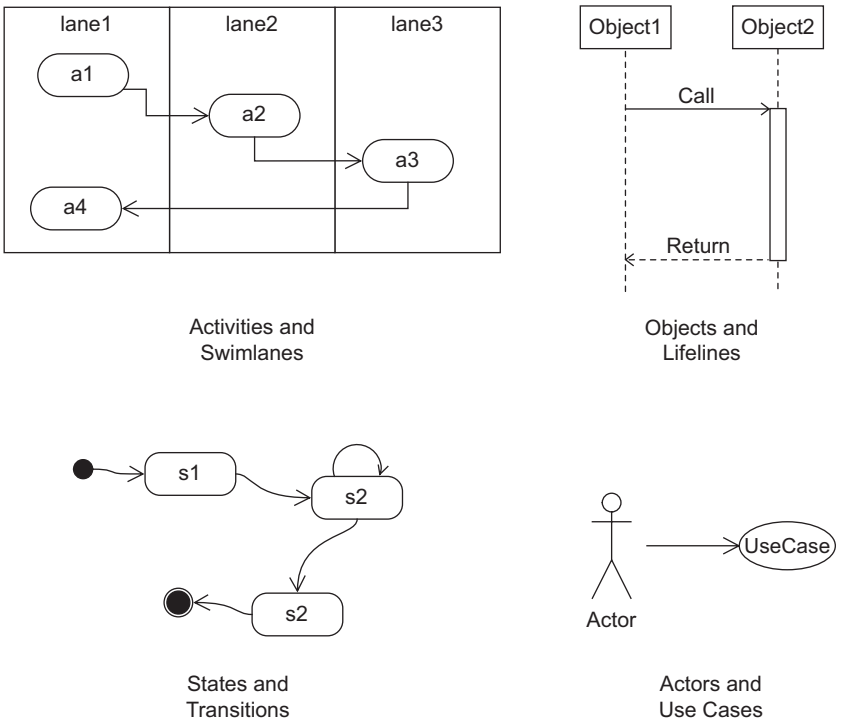

Figure 1-12: Behavioral conventions

Building your own graphical language on top of a given set of notational elements and conventions is analogous to building an embedded textual DSL, where instead of writing type wrappers and methods to make the language convenient to your domain, you define a mapping from the notational elements to your own domain concepts. If you want to define a graphical language that uses different notational elements and conventions, you have to be more expert and know how to create new diagrammatic elements from lower-level constructs. This is analogous to building your own parser for a textual DSL.

Aspects of Graphical DSLs

A graphical DSL has several important aspects that must be defined. The most important of these are its notation, domain model, generation, serialization, and tool integration.

Notation

In the previous section we talked about the notation of the language and how it can be built by reusing basic elements, often derived from well-established conventions, particularly those that originate in UML. For the kinds of graphical DSLs that we support, the basic building blocks are various kinds of shapes and connectors laid out on a two-dimensional drawing surface. These shapes and connectors contain decorators, which are used to display additional information such as text and icons attached to the shapes and connectors in particular places. In Chapter 4 we'll see full details of how to define these shapes and connectors and how to associate them with the other aspects of the language.

Domain Model

The domain model is a model of the concepts described by a language. The domain model for a graphical language plays a rather similar role in its definition to that played by a BNF grammar for a textual language. But for graphical languages, the domain model is usually itself represented graphically.

The basic building blocks for a domain model are *domain classes* and *domain relationships*. Each domain class represents a concept from the domain; each domain relationship represents a relationship between domain concepts. Typically, domain concepts are mapped to shapes in order to be represented on diagrams. Domain relationships can be mapped to connectors between those shapes or to physical relationships between shapes, such as containment.

Another important aspect of the domain model is the definition of *constraints*, which can be defined to check that diagrams created using the language are valid. For example, the class diagram in Figure 1-13 uses the correct diagrammatical conventions but defines a cyclic class hierarchy that is semantically invalid. Chapter 7 describes how to define constraints in the DSL Tools and discusses the differences between hard and soft constraints.

Generation

You define a language because you want to do something useful with it. Having created some models using the language, you normally want to generate some *artifacts*: some code, or data, or a configuration file, or another diagram, or even a combination of all of these. You'll want to be

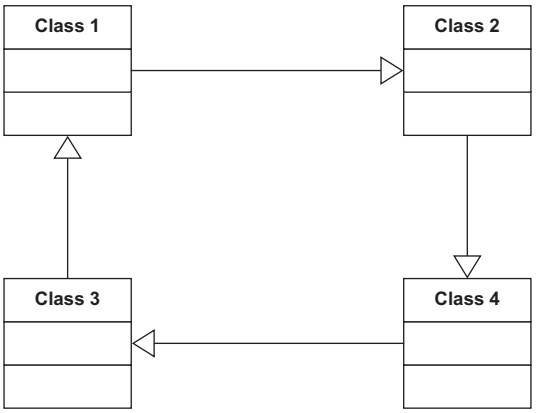

FIGURE 1-13: Invalid class diagram

able to regenerate these artifacts efficiently whenever you change a diagram, causing them to be checked out of source control if necessary.

Chapter 8 explains the DSL Tools generation framework, which enables the language author to define how to map models into useful artifacts.

Serialization

Having created some models, you'll want to save them, check them into source control, and reload them later. The information to save includes details about the shapes and connectors on the design surface, where they are positioned, and what color they are, as well as details of the domain concepts represented by those shapes.

It's often useful to be able to customize the XML format for saving models in order to help with integrating these models with other tools. This flexibility increases interoperability between tools and also makes it possible to use standard XML tools to manage and make changes to the saved models. Using an XML format that is easy to read also helps with source control conflicts. It is relatively straightforward to identify differences in versions of an artifact using textual differencing tools and to merge changes to artifacts successfully at the level of the XML files.

Chapter 6 explains how to define and customize the serialization format for a graphical DSL.

Tool Integration

The next important aspect of a graphical DSL design is to define how it will show up in the Visual Studio environment. This involves answering questions such as:

- What file extensions are associated with the language?
- When a file is opened, which windows appear, and what is the scope within Visual Studio of the information that is represented?
- Does the language have a tree-structured explorer, and if so, what do the nodes look like—with icons and/or strings—and how are they organized?
- How do the properties of selected elements appear in the properties browser?
- Are any custom editors designed for particular language elements?
- What icons appear on the toolbox when the diagram is being edited, and what happens when they are dragged and dropped?
- Which menu commands are enabled for different elements in the diagram and the associated windows, and what do they do?
- What happens if you double-click on a shape or connector?

Chapters 4, 5, and 10 describe how to define these behaviors and show ways of customizing the designer by adding your own code.

Putting It All Together

From the previous sections you can see that there are a lot of aspects to defining a DSL. This might seem rather daunting. Thankfully, the DSL Tools make it easier than you might think. Many of the aspects are created for you automatically, and you only need to worry about them if you want to change the way that they work. Complete languages are provided as starting points so that you don't need to start from scratch. Having defined your DSL, the DSL Tools are also used to generate code and artifacts that implement, test, and deploy the DSL as a designer fully integrated into Visual Studio. If you want to step outside of the set of features easily supported by the DSL Tools, we've provided many code customization options for that purpose.

The DSL Tools have even been used to define and build themselves. The DSL designer that is used to define domain models and notations is itself a DSL. Just like a compiler that can be used to compile itself, the DSL designer was used to define and generate itself.

DSLs in Visual Studio

Visual Studio 2005 has several graphical domain-specific languages integrated into it. These are the Distributed System Designers, which come with Visual Studio 2005 Team Edition for Software Architects, and the Class Designer which comes with Visual Studio 2005 Standard Edition and later. These designers are built on an earlier version of the DSL Tools; the current version is based on this earlier version and has evolved separately. The two versions are incompatible, which means that the DSL Tools cannot be used to extend the integrated designers.

Nevertheless, these designers illustrate very well some of the motivations for using domain-specific languages. Let's look at a simple example, using the Application Designer. This is a tool for modeling applications in distributed systems, with a particular emphasis on the endpoints that the applications implement and use, so that the user can wire the applications together into more complex systems. Figure 1-14 shows a simple design consisting of a Windows application, called **InvestmentCalculator**, that talks to an endpoint called **StockPrices**, which is implemented as a web service by an ASP.NET web application called **StockPriceApplication**. The **StockPrices** web service is shown as a *provider endpoint* on the **StockPriceApplication** node and is wired to a corresponding *consumer endpoint* on the **InvestmentCalculator** node.

Having created this design and chosen the implementation language, the Application Designer can generate the skeleton of an implementation for it using standard code templates installed with Visual Studio. The diagram context menu item "Implement All Applications …" causes the generation of two new projects in the solution, including the files needed to implement the solution, as shown in Figure 1-15. Implementing the application by generating these files like this requires much less work than does creating these files by hand. This is one clear benefit of defining a DSL—we can more quickly generate code that would be tedious and error-prone to write by hand.

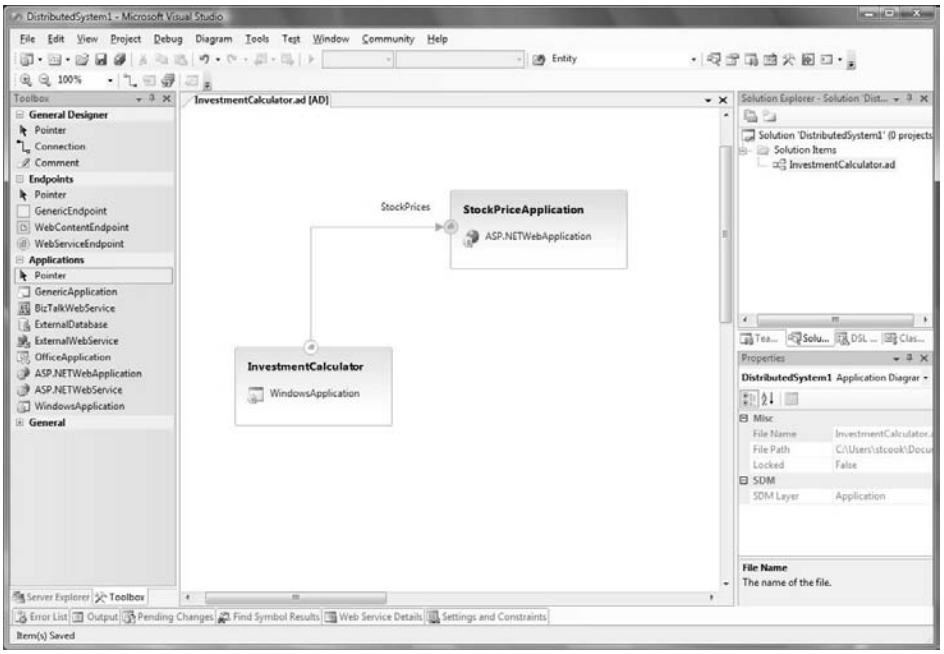

FIGURE 1-14: An application diagram

It's interesting to look into this solution and see where the name of the web service—**StockPrices**—appears. There are several places, in fact, including:

1. The name of the file `StockPrices.cs`.
2. The body of the generated file `StockPrices.cs`, containing the following code, which mentions `StockPrices` as the name of the class in the `Name` parameter of the `WebServiceBinding` attribute and in the `Binding` parameter of the `SoapDocumentMethod` attribute.

```
namespace StockPriceApplication
{
  [System.Web.Services.WebServiceBinding(Name = "StockPrices",
     ConformsTo = System.Web.Services.WsiProfiles.BasicProfile1_1,
     EmitConformanceClaims = true),
   System.Web.Services.Protocols.SoapDocumentService()]
  public class StockPrices : System.Web.Services.WebService
  {
    [System.Web.Services.WebMethod(),
     System.Web.Services.Protocols.SoapDocumentMethod(Binding="StockPrices")]
```

```
    public string GetPrice(string Symbol)
    {
      throw new System.NotImplementedException();
    }
  }
}
```

3. The name of the file `StockPrices.asmx`.
4. The body of the file `StockPrices.asmx`, containing the following template, which mentions `StockPrices` as a class name and a file name.

```
<%@ webservice class="StockPriceApplication.StockPrices"
               language="c#"
               codebehind="~/App_Code/StockPrices.cs" %>
```

5. The two SDM (System Definition Model) files. These are XML files that describe operational requirements for the applications and can be used to match these requirements against the operational facilities provided by a data center. This is not the place to go into the details of these files; suffice it to say that they both contain references to the service called **StockPrices**.
6. The web reference in the InvestmentCalculator application, which contains a URL such as http://localhost:2523/StockPriceApplication/StockPrices.asmx?wsdl.
7. The `app.config` file for the InvestmentCalculator application, containing the following section, which includes a reference to the filename `StockPrices.asmx` as well as the name **StockPrices** embedded in the longer name for the setting.

```
<applicationSettings>
  <InvestmentCalculator.Properties.Settings>
    <setting name="InvestmentCalculator_localhost_StockPrices"
                  serializeAs="String">
      <value>
          http://localhost:2523/StockPriceApplication/StockPrices.asmx
      </value>
    </setting>
  </InvestmentCalculator.Properties.Settings>
</applicationSettings>
```

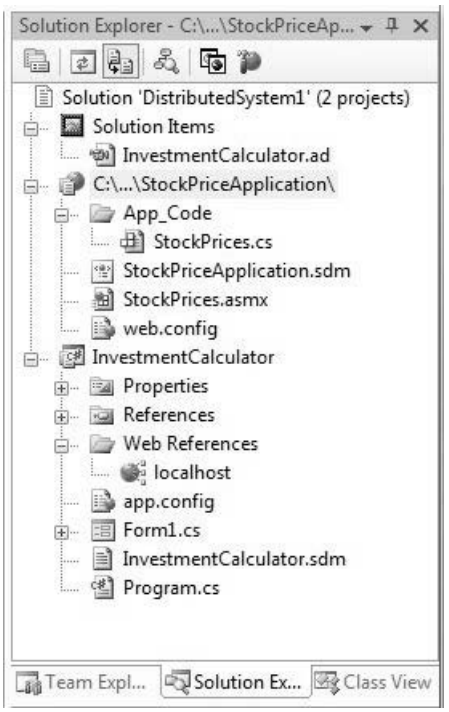

FIGURE 1-15: The generated solution

Now imagine that you want to change the name of the web service. Instead of **StockPrices,** you'd prefer to call it **StockValues**. Working in a modern coding environment, this should be a simple refactoring operation, such as the ones available from the "Refactor" menu in the code editor. But unfortunately, opening the `StockPrices.cs` file and using the "Refactor" menu will not have the desired effect, because many of the occurrences of the name **StockPrices** are not in code.

However, changing the name from **StockPrices** to **StockValues** on the Application Designer diagram does have the right effect. All of the references within the **StockPriceApplication** project are updated immediately, including the filenames and all of the references in the list above. At this point, the consumer endpoint on the **InvestmentCalculator** project is marked with a small warning symbol to indicate that it is referring to something that has changed; the web reference in the **InvestmentCalculator** project has been removed, and the `app.config` file no longer contains any reference to **StockPrices**. Selecting the "Implement" option from the context menu on the endpoint causes the web reference, `app.config`, and SDM

files to refer to the new name. By using the DSL, the operation of changing the name has been reduced from a time-consuming and error-prone combination of multiple manual edits to a simple two-step procedure carried out at the appropriate level of abstraction.

You may ask what happens if you change the name of **StockPrices** in just one of these generated artifacts. Well, by doing that you have invalidated your solution. In general, it is difficult or impossible for a tool to solve all of the possible round-tripping conundrums that could be created if you allow complete freedom to edit any artifact at any time. In this particular case, you are allowed to insert your own code into the body of the `GetPrice()` method, and that code will be preserved if the endpoint or operation name is changed in the model. But if you manually change the name of the class or method itself in the code, you have effectively broken the relationship between the code and the model, and future changes will not be synchronized. We return to the general problem of keeping models and artifacts synchronized in Chapter 8.

We can summarize the qualities of the Application Designer, which are qualities that any well-designed DSL should possess, as follows:

- It is a sharply focused tool for a specific task.
- The model corresponds closely to the domain being modeled, and the transformations required to generate code and other artifacts are simple.
- Because of these simple transformations, the round-tripping problem becomes tractable.
- The artifacts associated with the language are all files and can be maintained in a source-control system, and the tool is engineered so that it works effectively in this environment.
- The interactive user experience on a modern computer is rapid and intuitive.
- The files manipulated by the tool are user-readable text files, using published formats based on XML.

The Customization Pit

Applying the simple DSL pattern can make it easy to create a solution to your problem as long as the solution can be expressed fully in the DSL. But what if you want to create something slightly different? If there are no other facilities available for modifying the solution, then you have a "customization pit" (Figure 1-16)—within the boundaries of what the DSL can express, things are easy and comfortable, but outside of those boundaries, things are difficult or impossible.

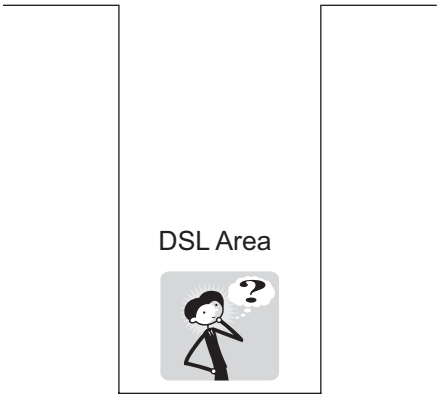

Figure 1-16: Customization pit

We'd much prefer the situation shown in Figure 1-17, where stepping out of the area covered by the DSL doesn't cause you to scale the walls of a deep pit but simply to step up onto a different plateau where things may be a little more difficult, but not impossibly hard. Beyond that plateau, there are further plateaus, each extending your capability to make solutions if you are willing and able to acquire the extra skills to go there. Alan Kay, the coinventor of Smalltalk, said, "Simple things should be simple. Complex things should be possible." We'd like to go a little further than that, and have difficulty increase only gradually as things get more complex.

There are several techniques that we can employ to achieve this. The first is to employ multiple DSLs, each one handling a different dimension of complexity in the problem, as depicted in Figure 1-18.

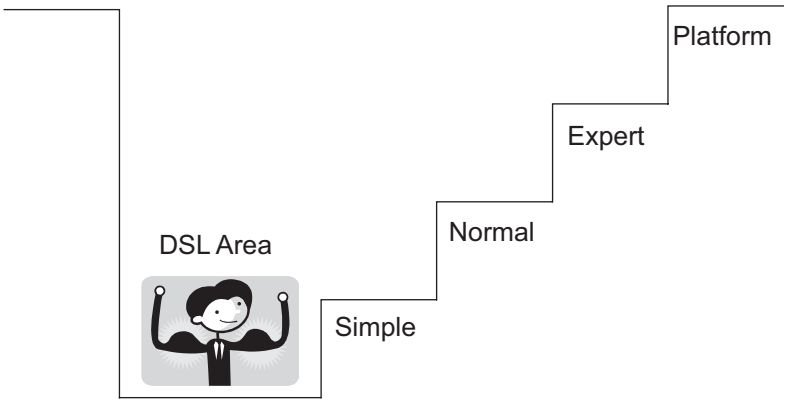

FIGURE 1-17: Customization staircase

A second technique, and one which we employ extensively in the design of the DSL Tools themselves, is to generate code that is explicitly designed to be extended. The C# 2.0 feature of partial classes is particularly helpful here, because part of a class can be generated while leaving other parts of the class to be written by hand. In the case of DSL Tools themselves, where the generated designer is hosted in Visual Studio, these code extensions can call upon facilities provided by the host, such as the user interface or the project system.

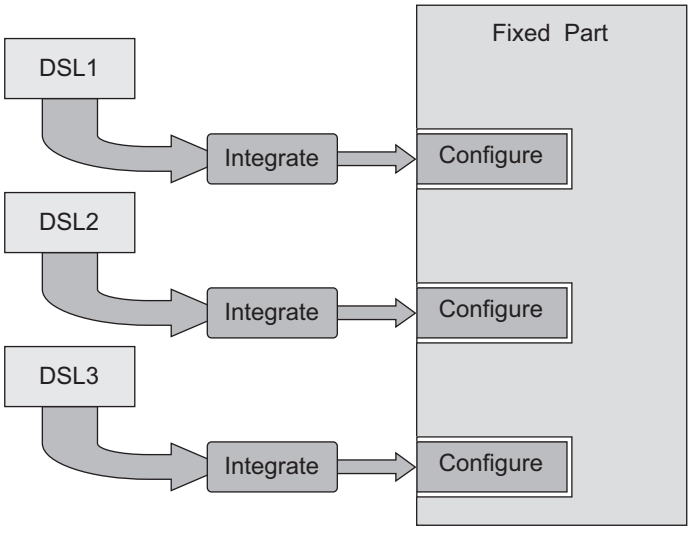

FIGURE 1-18: Multiple DSLs

A third technique, which you might think of as stepping up onto a higher-level *expert* plateau, is to enable the developer to modify the code-generation mechanisms, thus changing the way that the DSL is integrated into its environment. This requires yet more skill, because making it work correctly requires deeper knowledge of the remainder of the code.

The final technique represented by the highest plateau is to alter the implementation of the supporting platform, because it simply isn't capable of supporting the required features.

UML

The Unified Modeling Language, or UML, was first published in 1997 by the Object Management Group. UML unified three earlier approaches for graphically depicting software systems: the Booch method, the Object Modeling Technique, and the Object-Oriented Software Engineering method. The advantage of the UML was that it provided a standard set of notational conventions for describing aspects of a software system. Before the UML was published, different authors used different graphical elements to mean the same thing. Three examples are shown in Figure 1-19. The method described in Grady Booch's 1990 book, *Object-Oriented Analysis and Design with Applications*, represented a class by a cloud; the OMT method described in the 1991 book, *Object-Oriented Modeling and Design,* by James Rumbaugh and his colleagues, represented a class by a rectangle; and the 1992 book, *Object-Oriented Software Engineering: A Use Case Driven Approach*, by Ivar Jacobson and his colleagues, advocated representing a class by a little circle and distinguished diagrammatically between *entity* classes, *controller* classes, and *interface* classes. Many other approaches also existed at that time. UML succeeded in eliminating this "Tower of Babel"—almost all competing diagramming approaches vanished rapidly from the marketplace when UML appeared.

On publication, UML became increasingly popular as a technique for documenting the early phases of software development, especially those using object-oriented technologies. Class diagrams, use case diagrams, and sequence diagrams were especially popular for documenting the results of object-oriented analysis and object-oriented design.

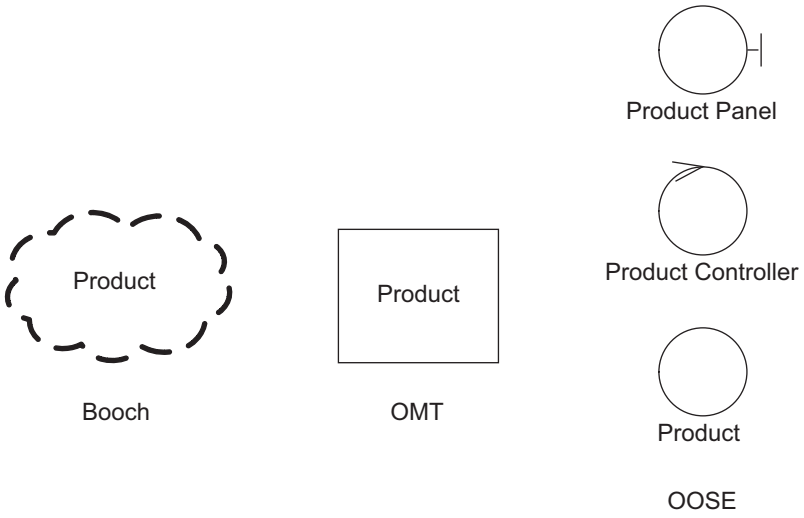

Figure 1-19: Different representations for a class

Figures 1-20 through Figure 1-22 show how to use UML to analyze the operation of a very simplified public library.

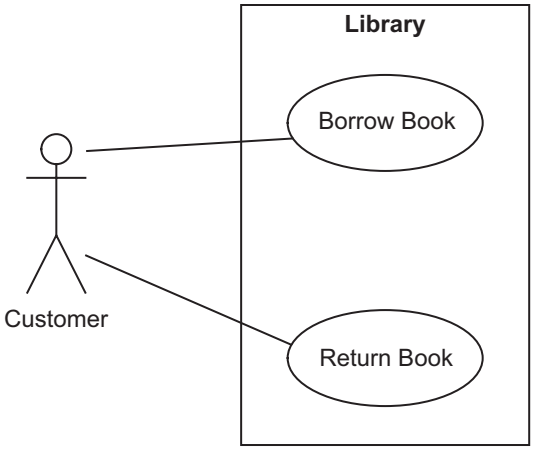

Figure 1-20: Use case diagram for simple library

The meaning of these diagrams is relatively informal. Being an analysis model, this set of diagrams does not exactly represent anything that happens in the software system. Instead, it helps the developer to make some

early decisions about what information will be represented in the software and how that information may be collected together and flow around when the system interacts with its environment. Translating the analysis model into an exact design for the actual software involves working out many details, such as the design of the database, the design of the classes that represent the business logic, the mapping between business logic and database classes, the design of the user interface, the messages that flow between clients and servers, and so on. Traces of the analysis model will be found in the design, but the detailed correspondence between the analysis model and the eventual programs, schemas, and definitions that constitute the running software will be complex.

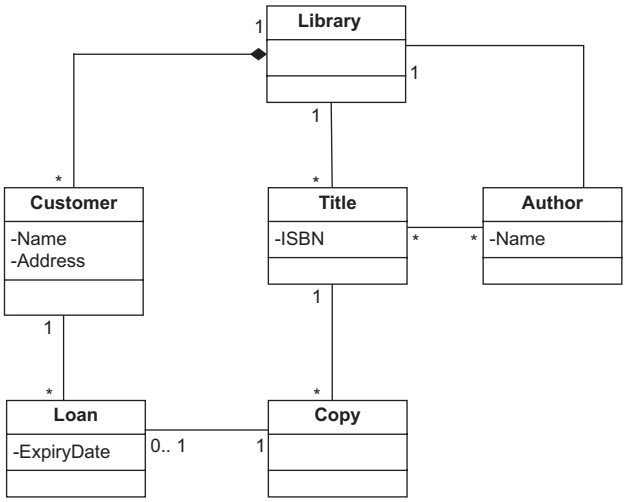

FIGURE 1-21: Class diagram for simple library

When UML emerged during the 1990s, mainstream thinking about object-oriented development assumed that there would be a relatively simple continuity between an object-oriented analysis and a corresponding object-oriented design. Several methodologies proposed that the way to get from the analysis to the design was simply to add detail while retaining the basic shape of the analysis. For simple examples, where there is a single computer implementing a simple non-distributed application, this can work, especially when no data persistence is involved.

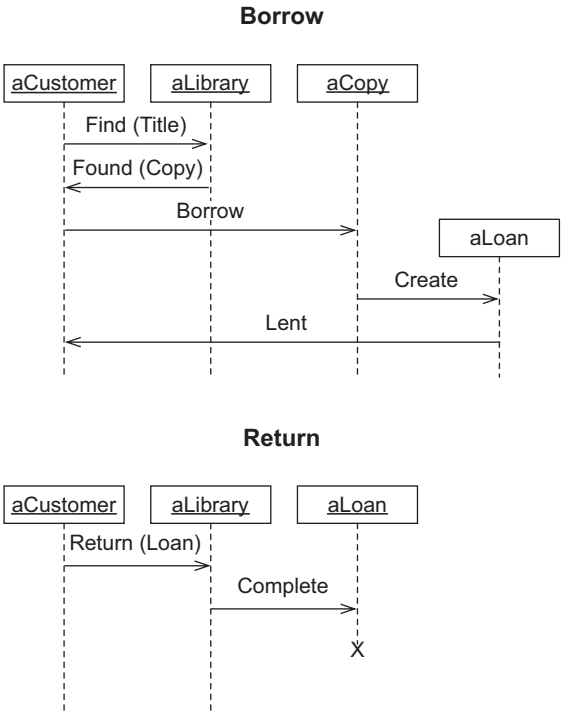

FIGURE 1-22: Sequence diagrams for simple library

The design of UML itself is actually based on this concept of adding implementation detail. The UML specification defines the ability to express the kind of detail found in an object-oriented programming language; for example, class members can be marked with the Java-inspired visibility values of `public`, `private`, `protected`, or `package`, and operations can have detailed signatures and so on. This helps to map a UML model to program code, especially if the programming language is Java. Note that there are many inconsistencies between the details of UML and Microsoft's Common Language Runtime, which make it more difficult to map UML effectively to the popular .NET languages Visual Basic and C#. When UML is used for a more abstract purpose such as analysis, these implementation details have to be ignored, because they are meaningless.

UML does offer limited extension facilities, called profiles, stereotypes, tagged values, and constraints. Stereotypes, tagged values, and constraints are mechanisms that add labels and restrictions to UML models to indicate

that a UML concept is being used to represent something else. So, for example, a UML class could be labeled as a «resource», or even as a «webpage»—the symbols «» are conventionally used to indicate that a stereotype is being used. But labeling a UML concept does not change anything else about it—a class still has attributes and operations, inheritance, and the rest of the built-in features.

A UML Profile is a packaged set of stereotypes, tagged values, and constraints that can be applied to a UML model. A tool can make use of the profile information to filter or hide elements but may not delete unwanted elements; a profile is effectively a viewing mechanism. These facilities do allow a limited amount of customization of UML for particular domains, and of course individual UML tool vendors can go beyond the published standard to provide increased levels of customization.

However, the world has moved on apace since UML was defined. The Internet and World Wide Web have matured, most of the computers in the world are connected together, and a multitude of new standards and technologies has emerged, especially XML and Web Services. In 2007 and beyond, the likely platform for implementing a business system will involve many distributed components executing in different computers. Logic and data are replicated for scalability and load balancing. Legacy systems are accessed on mainframes and servers. Firewalls and routers are configured to maintain security and connectivity. Browsers and smart clients are distributed to many different devices and appliances. Common artifacts in this world, such as Web Service Definition Language (WSDL) or configuration files, have no standard representations in UML. Although stereotypes and profiles can be used to apply UML in domains for which it was not designed, such an approach gives cumbersome results. In such a world, the transformation from a simple object-oriented analysis to a detailed system design is far too complex to be thought of simply as "adding detail." Different approaches are needed.

If UML is not convenient to be used directly, what happens if we open up the definition of UML, remove all of the parts we don't need, add new parts that we do need, and design a language specifically tailored for the generation task that we want to accomplish? In short, what would happen if we had an environment for constructing and manipulating graphical

modeling languages? The answer is that we would eliminate the mismatches and conceptual gaps that occur when we use a fixed modeling language, and we would make our development process more seamless and more efficient. That is the approach adopted in DSL Tools.

Instead of thinking about UML as a single language, we prefer to think of it as a set of reusable diagrammatic conventions, each of which can be applied to a particular kind of situation that we might encounter during software development. For example, sequence charts such as those in Figure 1-22 might be used to describe the flow of messages between applications in a distributed system, the flow of invocations between objects in an application, or even information interchange between departments in an organization. In the first case, the vertical lines on the diagram represent applications, in the second case they represent objects, and in the third case they represent departments.

Note also that it is not only end users that benefit from clean domain-specific abstractions. Developers who build tools that generate code and other artifacts from models and keep models coordinated with one another, need to access model data; providing APIs that work directly in terms of the abstractions of the problem domain is critical to productivity for developers. Developers want the API for the logical data center to give them direct access to the properties of an IIS server or a SQL Server database. Similarly, they want the API for the sequence charts to talk directly about applications, objects, or departments. They'd like to write strongly typed code, such as this:

```
foreach (Department dept in message.Receiver.SubDepartments)
{
  // generate some artifacts
}
```

This contrasts with having to reinterpret a model intended for other purposes (such as a UML model), which can give rise to code like this:

```
Lifeline lifeline = message.Receiver;
if (lifeline.Object.Label = "Department")
{
  Department receiver = lifeline.Object.Element as Department;
  if (receiver != null)
  {
    foreach (Department dept in receiver.SubDepartments)
```

```
    {
        // generate some artifacts
    }
  }
}
else
{
  // handle errors
}
```

SUMMARY

In this chapter we introduced Domain-Specific Development and discussed some examples and benefits of the approach.

We looked at how to define textual domain-specific languages as new languages or as embedded languages within an existing host language, and we saw how XML can be used as a simple and cost-effective substrate for defining textual DSLs. We discussed the different aspects of graphical DSLs, and saw how these are being implemented in several components of Visual Studio 2005. We talked about the customization pit and how to overcome it.

Finally, we discussed UML and saw how it provides a very popular set of conventions for creating diagrammatic documentation of software and how a domain-specific approach helps to overcome its disadvantages.