2883_Ch02_FINAL 3/26/04 9:48 AM Page 11

CHAPTER 2

Dissection of an Application Framework

WE LOOKED BRIEFLY in the previous chapter at what an application framework is, and we considered its benefits and costs. In this chapter, we will dive more deeply into the details of the framework. We will look specifically at what is in a framework, how we can develop a framework for our application, and what object-oriented techniques we can leverage in developing the framework.

To better understand how we can develop an application framework, we need first to understand what goes in an application framework and its relationship to other parts of the system.

Framework Layers

You learned in chapter one that an application framework is a "semifinished" application that can act as a starting point for a business application. Applications that are built on top of the framework consist of two layers: the application layer and the framework layer. The framework layer may consist of numerous components, which can be again grouped into domain-specific components and cross-domain components. Figure 2-1 illustrates the different participants in an application and their relationship to each other.

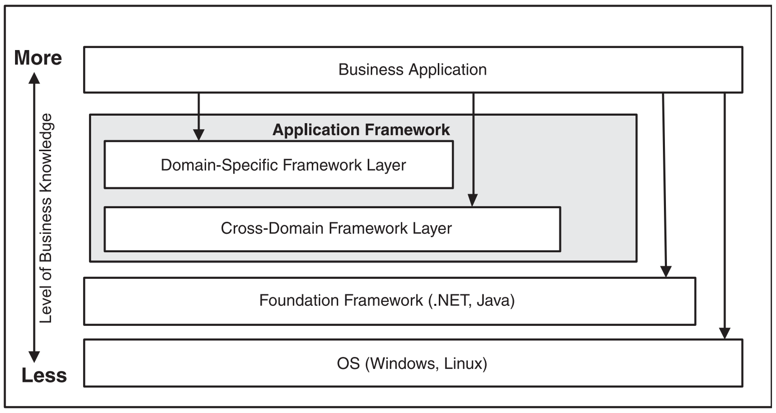
More
Level of Business Knowledge
Less
Business Application
Application Framework
Domain-Specific Framework Layer
Cross-Domain Framework Layer
Foundation Framework (.NET, Java)
OS (Windows, Linux)

Figure 2-1. Multiple layers within an application

The following is a brief description of what each layer represents and what role it plays in the overall system.

The Business Application

The business application represents the custom application that developers are responsible for. It implements the detailed business knowledge for the specific application under development. Developers build the business application according to the particular scenario described by business analysts. As the business logic and rules change, it is this level at which changes will mostly likely occur, particularly when such changes are minor and isolated.

The Application Framework

The application framework represents the semifinished application that architects have developed as a basis for developers to use to construct their business applications. The application framework can be broken down into two layers: a domain-specific framework layer and a cross-domain framework layer.

The Domain-Specific Framework Layer

The domain-specific framework consists of specialized framework components that target a specific business domain. In comparison to the business application layer, the domain-specific framework layer implements knowledge that is common to all applications of a particular business domain, in contrast to the business application layer, where the business knowledge and logic are targeted to a particular application.

You can think of a domain-specific framework as corresponding to a country's constitution and a business application as analogous to the laws of a particular state or local government. The constitution doesn't describe the specific laws that the state has to implement, but instead it describes the principles under which the system of laws should be framed. Each state may pass its own laws, but all those laws must be based on the principles set out in the constitution. However, as long as the state law is in conformity with the constitution, the state is free to create laws that are best suited to that state. Like the constitution, a domain-specific framework doesn't mandate how each business application should be built; instead, it provides a set of components that encapsulate the core business characteristics and processes of a particular business domain. For example, a shopping cart component describing a customer's selected product items, the quantity of each, and the time of selection can be considered a domain-specific framework component for the on-line B2C business domain. Different business applications (an on-line shopping site in this case) may use the shopping cart component differently in separate scenarios, but they all share a common trait: They all need an object that provides information on the customer's product selection, quantity of each selection, and time of the selection.

Unlike the business application, where developers are in charge of design and implementation of the actual application, the domain-specific framework layer is designed and implemented by persons who have expertise and deep understanding of a specific business area and know how to encapsulate and abstract business-domain knowledge in a form that can be easily adapted by developers in building the actual business application. Although software architecting skill is important in developing a business-specific framework, the business expertise is especially critical in the success of this layer of the framework.

The domain-specific framework layer contains business-domain knowledge that is much less volatile than that in the business applications and it expects few or no changes as the business rules change throughout the application.

The Cross-Domain Framework Layer

The cross-domain framework represents framework components that contain no business-domain knowledge. Because the business-domain knowledge is absent from this layer of the framework, it can be shared among multiple applications in different business domains. In other words, this layer hosts the components and services that are commonly found in most applications, regardless of their business domain.

There are many common themes among applications of different business domains that we can "package" into the cross-domain framework layer. For example, every application needs some way of managing the configuration information used by different parts of the application. A configuration service and architecture greatly reduces the development effort of numerous applications, regardless of the business domain of the application. In a distributed application environment, one application often needs to talk to another application residing on a different system, so an event notification service will also benefit the development effort of such applications by presenting a ready-to-use system that transmits information among the different applications. As you can see, if we can identify the common themes among different types of applications and develop services and components to take care of such common requirements, we can significantly increase code and design reuse throughout the applications.

Those who develop cross-domain framework layers are individuals who have developed a large number of applications and have a good understanding of software design as well as a knowledge of the features that are common to many applications. These individuals don't have to know a great deal about particular businesses, but they must have good object-oriented skills so that they can build the framework in such way that application developers can easily plug in their custom business logic to solve application-specific problems.

Since the cross-domain framework layer contains no specific business-domain knowledge, it may be thought of as generic to most applications. It is thus unaffected by changes to business rules and requirements. However, this layer will be affected by the recognition of new common themes that arise during the development process, for such themes will be implemented in this layer. Moreover, if it turns out that certain aspects of the framework's design interfere with the adaptation of the business application, the cross-domain framework will have to be modified.

The Foundation Framework

The foundation framework represents the programming model on which the application framework and business application are built. This layer is developed by the software vendor. Some of the best-known foundation frameworks are Sun's Java environment and Microsoft's .NET Framework. The foundation framework is used to develop a wide variety of applications, and it contains no specific business-domain knowledge. Changes to the foundation framework are driven primarily by the need for higher performance or the support of newer technologies.

OS

The OS layer represents, as its name implies, the operating system level. It provides access to system resources, such as CPU, memory, and disks, and to all the layers that sit on top of it.

With a basic understanding of the different layers in the overall picture and how they are positioned in an application, let's take a look at how to actually build an application framework. First, we will look at the application framework's development process.

The Framework Development Process

After you have decided that you need an application framework, you should first determine the major phases involved in a framework development process. Figure 2-2 shows four majors phases: analysis, design, development, and stabilization.

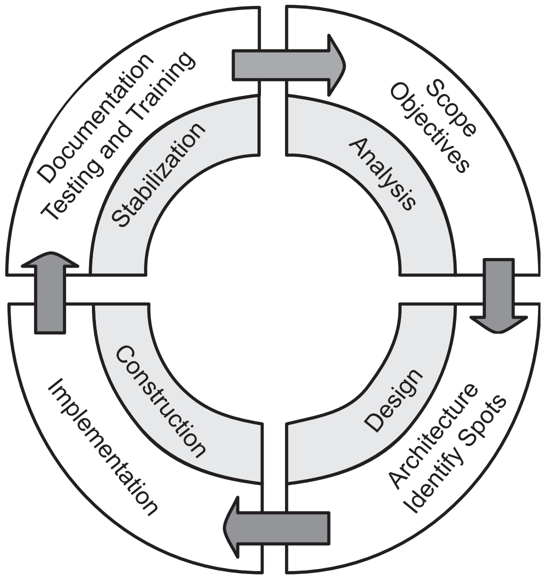

Figure 2-2. Framework development process

The gray inner circle indicates the phase, and the outer circle indicates some of the major tasks involved in each phase. Let's take a look at what is involved in each phase.

Analysis

As we start the process of developing an application framework, the first phase we enter is the analysis phase. As with application development, framework development starts by first setting the scope and objectives of the project or framework. We need first to identify the key features that are to be included in the framework. What types of business applications will be relying on this framework? What use cases will the framework support? In other words, how will developers be able to develop their business applications on top of this framework? The framework is built to support the development of the business application, so it is important to figure out what business domains the framework will able to support. Many questions will be asked during the analysis phase of framework development to set the scope and objectives of the framework.

During the analysis phase, we also need to create an iteration plan for improving the framework over time. Framework development involves complex and abstract tasks, and you shouldn't expect to get everything right on the first iteration. You may discover that certain items need to be added to or removed from the framework as you start implementing it. You may also decide to modify how some parts of the framework work to help developers become more productive in adapting the framework. You need to set up a plan to collect ideas for enhancements and fixes as developers start using the framework for use as the input to the next iteration. In addition to such an iteration plan, you will also need to draft a project plan and establish a timeline and documentation of major milestones for all phases of the process.

Design

After we have set the objectives for the application framework, the next phase is the design phase. The design phase for framework development involves two major tasks. First, we need to identity the common spots and the hot spots in both the domain-specific layer and the cross-domain framework layer. Second, we need to devise an architecture for the framework that will be used as a blueprint during the construction phase.

"Common spot" and "hot spot" are special terms relating to framework development. You will learn more about them later in the chapter. In a nutshell, a common spot is an area in the framework where variation is unlikely. A common spot is often a framework component or service that is ready for use without significant customization by application developers. On the other hand, a hot spot is an area in the framework where variation is frequent. Developers must provide their specific business logic in those hot spots in order to use the framework component. A hot spot is an abstract method that requires the developer to implement specific business logic. Identifying the common spots and hot spots in a business application allows you to identify the specific components and services in the framework. Identifying what is variable and what is fixed in a business domain is not an easy task. The business experts and software architects need to work together to identify those spots among the different layers of the framework in order to design a framework that is both easy to use and extensible.

After the business experts and software architects have come up with a list of components and services and identify which are common spots and which hot spots, the software architects can start designing the blueprint of the framework. As part of the architectural design of the framework, you need to create a number of design deliverables, such as class diagrams and activity diagrams, which will be used during the construction phase of the framework. Software architects also spend their time thinking about the techniques they can use on various components and services, such as design patterns, to maximize code reuse and extensibility in the final framework.

During the design phase, you can also begin to create a prototype of the application framework and then build a sample application on top of it. Testing the prototype with a sample application helps you learn how the framework you develop will be used to build the business application and gain insight into potential improvements in the design of the framework.

Implementation

After the framework design is done, the next phase is to actually code the application framework. The implementation phase has one goal, which is to develop a framework that meets the requirements and time constraints. If you have done a good design job on the framework, implementing it is no different from regular application development. As with application development, where different team members can work on different parts of the same application simultaneously, you can have work progressing on different pieces of the application framework simultaneously, as long as the framework is partitioned into well-defined modules. Unit testing on the newly created functionality can be tricky during framework development. In the case of application development, developers will simply test the use case on the newly created functionality to see whether it works as desired. However, unit testing of frameworks is much less direct and visual than that of business applications, since the application that uses the framework simply hasn't been created yet and all the business data needed by the application have not been generated.

As with application development, you can create number of milestone releases during the implementation phase. You can create several incremental releases so that you can get part of the application framework to developers and start getting feedback on any potential improvements and fixes for the next release.

Stabilization

The stabilization phase, the last phase before the next iteration starts, focuses on testing, bug fixing, developer feedback, documentation, and knowledge transfer.

Testing the framework in this phase is often driven by developers instead of professional quality-assurance testers or business end users, since developers are the primary users of the framework. During the stabilization phase of the framework development, the design and construction of the actual application have not yet started, so developers haven't yet created much application code. In such a situation, you need to identity at least one usage scenario in the business application for each framework component and service and ask the development team to write a small portion of the application based on such usage scenarios. Although it is not possible to test every framework usage scenario of an application that has not yet been built, you can effectively test your framework through the selective implementation of part of a business application that is representative of the usage pattern of the framework.

Of course, in order for developers to develop part of an application based on the framework, they must know how to use the framework productively. As the creator of the application framework, you will initiate developer training in the stabilization phase. The better job you do educating developers about the framework, the better the developers will be able to test your framework and the more productive they will be when they start using your framework extensively in application development.

Besides conducting frequent training sessions on how to use the application framework, you also need to produce good documentation of your framework, which will be an invaluable tool in helping developers learn to use the framework.

Typical documentation created to help developers consists of four parts:

- An overview of the framework that explains the purpose of the framework and the major components and services available in the framework.
- Some pictures, diagrams, and descriptions of the framework that help developers grasp the framework and its design philosophy.
- An API reference for the functionalities inside the framework that enables developers to look up the framework's functionalities during the development process.
- A collection of examples that show how the framework is used. Concrete examples of the framework in action best demonstrate the usage scenario of the application framework and help shorten the learning process for developers.

The stabilization phase is also where the framework designer will continually collect developers' feedback on the framework for the next iteration of framework development. After developers start using the framework, the framework designer must also participate in the effort of application development by providing assistance to the developer team, answering questions and solving problems related to the usability of the framework.

With a development process in place for framework development, let's look at some common approaches and techniques for developing a framework that we will use in the rest of the book

Framework Development Techniques

In order to develop an effective application framework, you need to know some common techniques for framework development. The following list shows some useful techniques and approaches that can help you develop a framework that is both easy to use and extensible.

- Common spots
- Hot spots
- Black-box framework
- White-box framework
- Gray-box framework
- Design patterns

Common spots, hot spots, and design patterns are some of the techniques used in framework development. Black, white, and gray boxes represent the approaches you can take to developing the framework.

Common Spots

Common spots, as the name suggests, represent the places in the business application where a common theme is repeated over and over again. If certain parts of the application repeat without much variation throughout the application, we can extract such common spots out of the application and package them into components in the framework layer. By moving such common spots into the framework, you avoid the duplication of such common spots throughout the application, and hence promote code reuse. This reduces the development effort when developers can simply referencing to the common spot in the framework from their applications. Figure 2-3 shows how the common spots relate to the framework and business application.

2883_Ch02_FINAL 3/26/04 9:48 AM Page 20

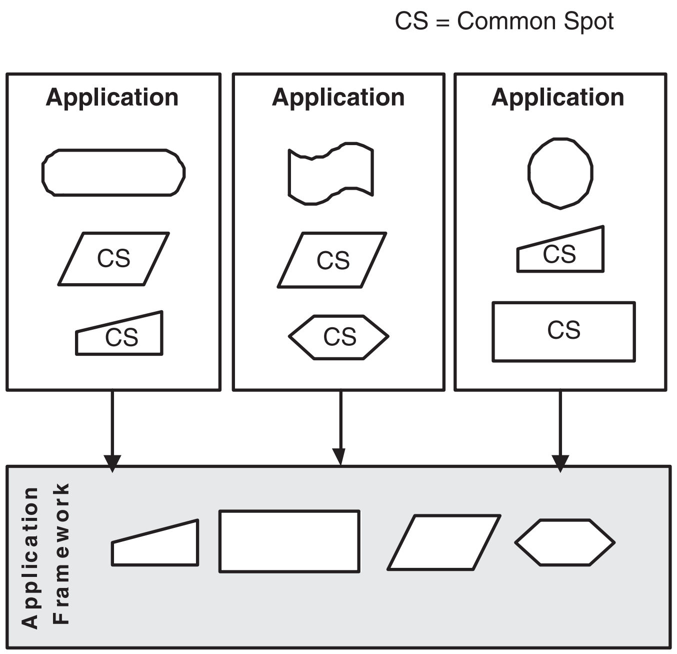

Figure 2-3. Common spots

In Figure 2-3, the application framework contains the components that provide the implementation for various common spots found in the application. When developers start building applications, they will reference the common spots implemented in the framework component instead of developing them themselves. As a result, the amount of application code they have to write is reduced.

To qualify as a common spot, a theme does not have appear in exactly the same way throughout the application. As long as the variations are small, you can still treat them as common spots and handle the small variations through parameterization and/or configuration settings.

The actual task of moving the common spot theme into a framework component is not hard. The difficult task is to identify in the analysis phase the common spots throughout the business application that have not yet been developed. It is not always easy to see through the common theme embedded in the application, and it usually takes a few tries to get it right.

Common spots can exist in both the domain-specific framework layer and the cross-domain framework layer. For example, the exchange of business documents is the central theme of B2B applications. A business document object would be considered one of the common themes that can be turned into a framework component for the domain-specific framework layer. Another example is a data cryptography service. Regardless of the type of application, data encryption and decryption are often applied at different parts of the application. A data encryption/decryption component, which simplifies and reduces the amount of code developers have to write to support cryptographic needs in the application, would be considered one of the components in the cross-domain framework layer.

From a technical point of view, implementation of common spots is straightforward. After identifying a common spot, the framework designer can develop the components that encapsulate the theme and logic in the common spot. Such components often take the form of concrete classes or executables. To accommodate the small variations in the component, some configuration data may also accompany the component. For example, you can allocate certain sections in the configuration file for parameterization of the framework components. In terms of adapting such framework components, developers need to write little or no code to use the component within the application.

Common spots capture the themes that are repeated throughout the application; however, each business application is unique, and there are as many spots where each application varies significantly due to the nature of the application as there are common spots. As a framework designer, you need to take account of those variations when designing the framework and make sure that developers can take advantage of the framework not only when there are common themes among applications, but also significant variation and customization between applications. This leads us to the next topic: hot spots.

Hot Spots

A hot spot is point of flexibility in the application framework. Another way to look at the hot spot is that it is a placeholder embedded in the framework where application-specific customization occurs.

Hot spots are the opposite of common spots. In a common spot, the framework implements the common themes inside the framework component; however, in the case of hot spots, there is nothing for the framework to implement except to leave an empty placeholder, which is later filled with a custom implementation by the business application built on top of the framework. Because each business application is responsible for providing its own implementation for the hot spot in the framework, the framework will behave differently for each business application. This is how a framework can be designed to suit different business applications in spite of significant variations among these business applications. Figure 2-4 illustrates how an application framework achieves flexibility through hot spots.

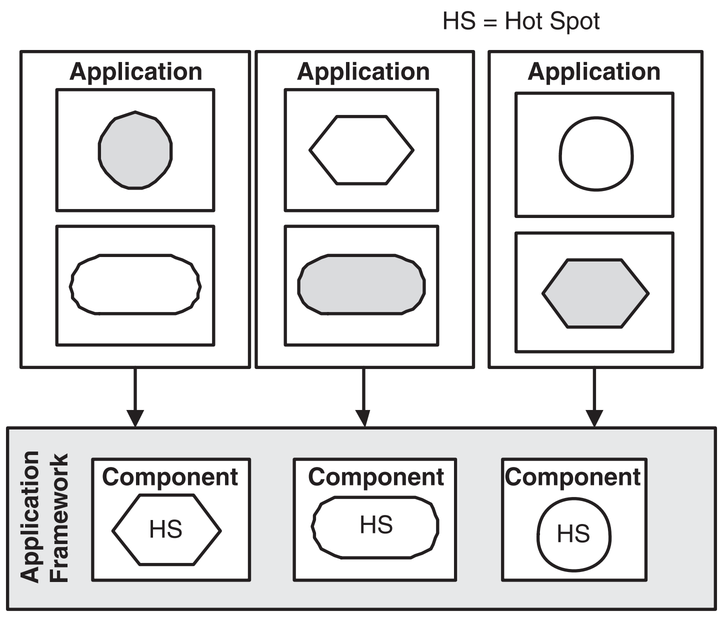

Figure 2-4. Hot spots

In Figure 2-4, components in the application framework layer consist of different hot spots, or empty placeholders for customization. Each application may use a number of framework components throughout the application. When the application uses a framework component that contains the hot spot, it needs to provide the implementation only in the hot spot in order to use the framework. The different designs and shading of the hot spot shapes shown in the figure represent the different implementations among various applications. As you can see, by implementing different logic inside the hot spots, each application will elicit different behaviors from the framework components.

As with common spots, identifying the hot spots may not be easy. To identify a potential hot spot, you must understand the business domain inside out and understand which points in the business application will potentially need to be customized. Having too many unnecessary hot spots in the framework will lead to extra coding effort for the developer team. Having too few hot spots makes it harder for the developer team to adapt the framework when desired customization becomes difficult due to the inflexibility of the underlying framework. Although it is possible to achieve such customization by overriding most or all the virtual methods in the base class, doing so would diminish code reuse and the purpose of inheritance.

Creating hot spots in the framework is not as straightforward as creating common spots. There are two approaches on how hot spots are enabled in the framework: the inheritance approach, and the composition approach. Let's look at the inheritance approach first.

The Inheritance Approach

The inheritance approach is driven by two important object-oriented concepts: hook methods and template methods.

Hook Methods

A hook method is a placeholder that will be filled by the application-specific logic. It is the manifestation of a hot spot concept in an actual class. The terms "hook method" and "hot spot" are interchangeable in many publications. However, a hot spot doesn't necessarily take the form of a method; it can also be in the form of a class or application, although the method form is the most common manifestation of a hot spot. Although a hook method can also appear in a concrete class (as we will see later in this section), it often appears in the form of an abstract method inside an abstract class.

An abstract class, by definition, is a class that contains abstract methods. An abstract method defined in the abstract class doesn't contain its method implementation. In order to use an abstract class, a class must inherit from the abstract class and implement the abstract methods in the parent class. Because of this feature, the derived class has the opportunity to inject some customized logic to make it behave according to some specific business requirement. The following example shows how abstract methods work:

```
public abstract class BasicBusiness
{
     protected float income;
     //the template method
     public void ReportTax()
     {
          float sTax = CalculateStateTax();
          float fTax = CalculateFedTax();
          bool ok = CheckBankBalance(sTax + fTax);
          if (!ok)
          {
               FileBankruptcy();
          }
```

2883_Ch02_FINAL 3/26/04 9:48 AM Page 24

```
          else
          {
               SendMoneyToGov(sTax + fTax);
          }

     }

     protected abstract float CalculateStateTax();
     protected abstract float CalculateFedTax();
}
```

The BasicBusiness class is an abstract class containing three methods: ReportTax, CalculateStateTax, and CalculateFedTax; the latter two are abstract methods. BasicBusiness is used to report income tax to state and federal government, and it contains the fictitious business-domain knowledge about the action taken at the time of tax filing. ReportTax determines whether the bank balance can cover the total tax. If the balance can cover it, a check will be sent out to the government; otherwise, the account holder will file for bankruptcy.

The ReportTax method depends on two pieces of information in order to determine whether to pay the tax or file for bankruptcy: the tax amount to be paid to the state government and the tax amount to be paid to the federal government. The creators of the BasicBusiness component have no knowledge of the tax law that applies to each situation. For example, depending on the location and type of business, different tax brackets may apply. Since BasicBusiness doesn't know how to calculate the tax amount, it will leave such tasks to someone who knows. The two abstract methods will act as placeholders to be filled later with code that generates the final tax amounts. Although CalculateStateTax and CalculateFedTax are abstract methods, they can be called just like regular methods. As you can see in the previous sample, the CheckBankBalance method takes the return values of two abstract methods as its parameters even though the two abstract methods have not yet been implemented.

Of course, someone has to implement the CalculateStateTax and CalculateFedTax abstract methods before the ReportTax method can provide any meaningful functionality. In fact, because BasicBusiness contains the abstract methods, it can't be used by instantiating it. Instead, you will need to create a concrete class that derives from the BasicBusiness class and implements its two abstract methods. The following example shows a class that extends BasicBusiness and implements its two abstract methods:

```
public class NewYorkBusiness : BasicBusiness
{
     //implementation of abstract method
     protected override float CalculateStateTax()
```

```
    {
        return income * 0.1F;
    }
    //implementation of abstract method
    protected override float CalculateFedTax()
    {
        return income * 0.2F;
    }
}
```

`NewYorkBusiness` is a concrete class that provides the tax calculation methods for New York State. With the custom implementation in the `NewYorkBusiness` class, `ReportTax` defined in `BasicBusiness` can perform some meaningful actions, as shown in the next example:

```
BasicBusiness nyBusiness = new NewYorkBusiness();
//ReportTax now will use the tax calculation algorithm defined for New York.
nyBusiness.ReportTax();

BasicBusiness caBusiness = new CaliforniaBusiness();
//ReportTax now will use the tax calculation algorithm defined for California.
caBusiness.ReportTax();
```

Abstract methods are a very powerful concept in both application development and framework development. Imagine that the `BasicBusiness` class is a framework component and the `NewYorkBusiness` is an application component. `BasicBusiness` leaves the hot spots (`CalculateStateTax` and `CalculateFedTax`) for the application to fill. Each application will fill those spots according to its specific requirements.

Although the application provides the implementation for the abstract method, it often doesn't invoke the abstract methods directly. Abstract methods are often invoked through the template method.

Template Methods

A template method, one of the GOF design patterns, describes a skeleton or process flow for certain operations rather than prescribing how each operation is carried out. In the example of our `BasicBusiness` component, `ReportTax` is a template method. The `ReportTax` method describes what steps are involved in reporting tax, but it does not describe how every step is performed, since some of the methods it references haven't been implemented. The template method emphasizes how the coordination between different objects/methods is carried out. In framework development, template methods contain the business-domain knowledge on how different methods should work together, whereas abstract methods provide a means of custom method implementation that is referenced in the template method. It is important to realize that template methods and hook methods embody two different concepts; Figure 2-5 illustrates the relationship between an abstract method (or hook method) and a template method in a framework.

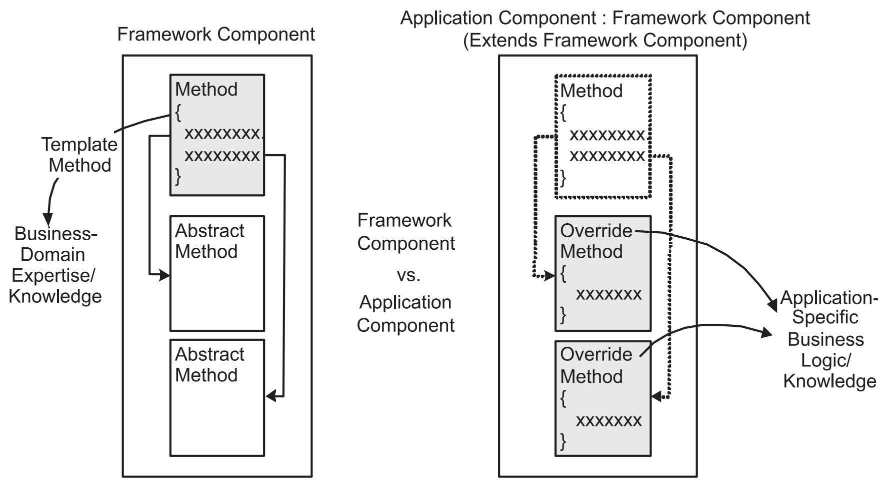
Framework Component
Application Component : Framework Component
(Extends Framework Component)
Template Method
Business-Domain Expertise/ Knowledge
Method { xxxxxxxx xxxxxxxx }
Abstract Method
Abstract Method
Framework Component vs. Application Component
Method { xxxxxxxx xxxxxxxx }
Override Method { xxxxxxx }
Override Method { xxxxxxx }
Application-Specific Business Logic/ Knowledge

Figure 2-5. Template method and abstract method

The left side of the figure represents a framework component, which contains the concrete template method that contains the business expertise and knowledge for a specific business domain. The framework component also has a number of hot spots, represented by the abstract methods. The arrows that point from the template method to the abstract methods indicate that the template method calls several abstract methods for their application-specific behaviors.

On the right side of the figure, an application component extends the framework component and implements the abstract methods in the framework component with the application-specific business logic and knowledge. Because the application framework inherits the functionalities of its framework component parent, its overridden methods (or hook methods) and the template method in its parent class together should be able to provide a combination of business-domain expertise/knowledge and application-level business logics.

Hot spots can be enabled not only through the hook and template methods as we just saw, but also through the concept of pluggable objects in a composition approach.

NOTE *A term has been used in several technical papers to describe this framework approach as the "Hollywood principle" or the "don't call me, I'll call you principle." Such terms describe the characteristic that a template method in the framework calls the application code instead of the application code calling the framework. However, this terminology is somewhat misleading. The reason such terms are used to describe a framework is that they stress that it is the template method in the framework that calls the implemented abstract methods and controls the process flow in the application. In most cases, the application always calls the framework. Although it may appear that a template method is calling the application code, it is the application code that first invokes the template method.*

The Composition Approach

The inheritance approach is a simple approach to enabling hot spots inside a framework, but because developers must know what data and methods are available inside the parent class as well as their interrelations in order to implement the abstract method, they often are required to have a very detailed knowledge of a framework in order to use it.

For example, in the `NewYorkBusiness` class, as easy as the implementation of `CalculateStateTax` and `CalculateFedTax` may seem, in order to implement these two methods, the developer must know that there is a protected float variable called `income` and that its value must have been set somewhere else prior to invoking either of the `CalculateXxxTax` methods. Also, exposing the internal details of the child class diminishes the encapsulation of the parent class, which can lead to the uncontrolled access and modification of the class's internal state by developers that are beyond the intent of parent class's designer. The following code segment shows the filling of the "hot spot" in an inheritance approach:

```
public class NewYorkBusiness : BasicBusiness
{
     protected override float CalculateStateTax()
     {
          return income * 0.1F;
        }
     protected override float CalculateFedTax()
     {
              return income * 0.2F;
        }
}
```

As you can imagine, as the method grows more complicated, developers may need to reference more data and methods inside the parent class and understand what the consequences are of setting the values of such data and calling such methods to change the overall state of the object.

Requiring developers to know such detailed information about the parent class stretches their learning curve for the framework and is burdensome to developers using the framework. One way to keep developers from having to learn the internal details of framework components is to define the hot spots as a set of interfaces that are well connected with the framework. Developers can create components by implementing such interfaces, which can then be plugged into the framework to customize its behaviors.

Pluggable Components

To enable hot spots through pluggable components, the application framework must first define the interface for the hot spot. An interface describes a set of methods a class must implement to be considered compatible with the interface. The interface describes what the methods inside the class look like, such as the method names, the number of parameters, and the parameter types, but not how they should be implemented. The following is an example of an interface definition in C#:

```
public interface ICalculateTax
{
     float CalculateStateTax();
     float CalculateFedTax();
     float Income
     {
         get;set;
        }
}
```

You can then create application components that support `ICalculateTax` by creating a concrete class and implementing every method/property defined in the interface, such as the `NewYorkBusiness` class shown as follows:

```
public class NewYorkBusiness : ICalculateTax
{
     private float income;
     public float Income
     {
          get { return income; }
```

```
            set { income = value; }
        }
        public float CalculateStateTax()
        {
            return income * 0.1F;
        }
        public float CalculateFedTax()
        {
            return income * 0.2F;
        }
    }
```

Because NewYorkBusiness implements the ICalculateTax interface, it now becomes compatible with or pluggable to any hot spot in the framework that can work with the ICalculateTax interface. With the help of interfaces, application developers can compose the custom behaviors into the underlying framework by loading the pluggable application components in the hot spots of the framework. Figure 2-6 illustrates how the composition approach enables the hot spots in the application framework.

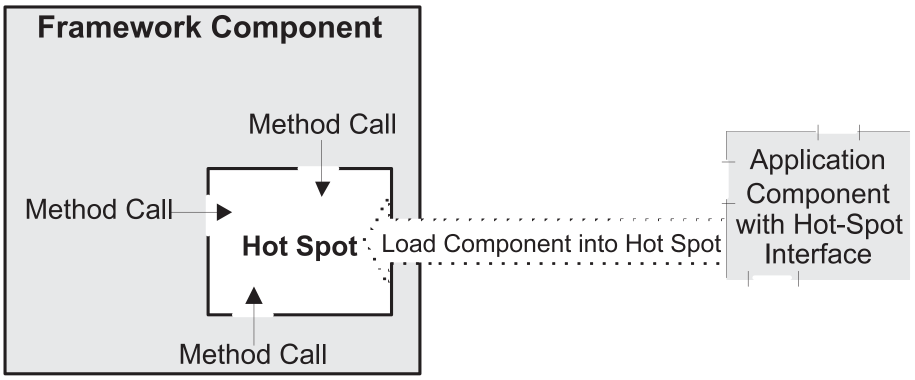
Framework Component
Method Call
Method Call
Hot Spot
Method Call
Load Component into Hot Spot
Application Component with Hot-Spot Interface

Figure 2-6. A pluggable application component

In using the composition approach to enable the hot spot, developers will need to create pluggable application components that have matching interfaces with the hot spot in the framework. The developer can then plug the component into the hot spot by binding the application component and the framework component together.

This composition approach for enabling hot spots is based on yet another GOF design pattern called “strategy.” You will learn more about the strategy pattern in Chapter 5.

Now let’s convert the tax example so that the hot spot is enabled through the composition approach. First, we need to modify the `BasicBusiness` component. Instead of making it an abstract class, this time we will make it a concrete class. The following code snippet shows the new `BasicBusiness` component:

```
public class BasicBusiness
{
     public void ReportTax (ICalculateTax calculator)
     {
          float sTax = calculator.CalculateStateTax();
          float fTax = calculator.CalculateFedTax();
          bool ok = CheckBankBalance(sTax + fTax);
          if (!ok)
          {
               FileBankruptcy();
          }
          else
          {
               SendMoneyToGov(sTax + fTax);
          }
     }
}
```

The `ReportTax` method now takes an input parameter of `ICalculateTax` type. This input parameter will provide the custom tax calculation mechanism, which is also a hot spot in the `BasicBusiness` framework component. As you can see, by plugging in the custom application component, we effectively “fill up” the hot spot with application-specific business logic/knowledge.

The following example shows how we can plug the application component into the framework from the application code:

```
ICalculateTax nyBusiness = new NewYorkBusiness();
ICalculateTax caBusiness = new CaliforniaBusiness();

BasicBusiness basic= new BasicBusiness();
basic.ReportTax(nyBusiness);
basic.ReportTax(caBusiness);
```

In the previous example, each `ReportTax` call will result in a different outcome, since the framework component is bound to a different `ICalculateTax` component on each call.

In order for the framework component to load the pluggable object, you can either use the approach described in the previous example or store the application component's type information inside a configuration file, then load the appropriate component through reflection and plug it right into the framework component dynamically.

Identifying and enabling common spots and hot spots are the central themes of application framework development. Depending on how you enable such spots, you create either a white-box framework, a black-box framework, or a gray-box framework.

White-Box Frameworks

A white-box framework is a framework that consists of abstract classes. Adapting a white-box framework requires developers to create concrete classes that inherit the abstract classes in the framework. White-box frameworks take on the inheritance approach to enable their hot spots. Figure 2-7 shows a white-box framework.

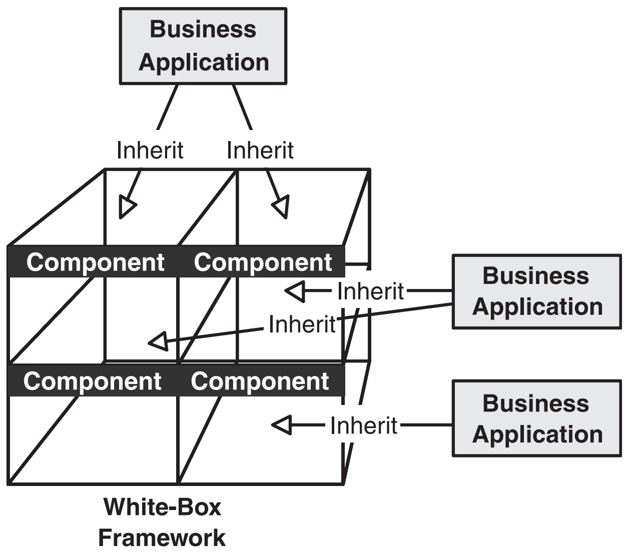

Figure 2-7. A white-box framework

A white-box framework is relatively easy to develop. You can start developing the abstract class by looking at some of the similar applications you have developed before, identifying their hot spots, and making them the abstract methods. When developing white-box frameworks, you are making an assumption about the pattern of process flow involved in each framework component through the template method. You often base these assumptions on business-domain expertise and prior business-application development experience. As developers start adapting the white-box framework, they need to program only a small number of “override” methods in the derived class and don’t have to worry about the overall process flow or how the abstract method is used inside the framework. This allows developers to focus solely on the abstract method without worrying about how the methods they are overriding relate to the rest of the framework.

Of course, there is a tradeoff in almost everything. White-box frameworks are very easy to design and develop, but they have their drawbacks. The first drawback is their inflexibility. In a white-box framework, when you determine how the process flow occurs inside a component through the template method you effectively hard-code the process flow and coordination logic in the component. Although developers who adapt the component may change the logic of the component’s hot spot, the overall process flow is nevertheless fixed. This “hard-coded” process flow is reflected as inflexibility when a change in business rules triggers a change in the process flow in the component. Because the process flow and coordination logic are fixed, you would have to update the existing component or write a new one that carries the process flow and coordination logic.

Another drawback of a white-box framework is that it often requires the developer to know many implementation details of the framework component. As the developer is implementing the abstract method in the framework component, he or she often needs to reference the abstract class’s methods and variables in the implementation code. This makes the understanding of internal details of the framework component an important prerequisite for correctly adapting the component.

A black-box framework often takes a composition-style approach to solving some of the challenges of the white box, but it has its own share of drawbacks.

Black-Box Frameworks

Black-box frameworks consist of concrete and ready-to-use classes and services. Although developers can extend the existing framework components to achieve customization in a black-box framework, they more often adapt the framework by combining a number of components to create the desired result. A black-box framework may contain many common spots, and it employs the composition approach to enable its hot spots. Figure 2-8 illustrates a black-box framework.

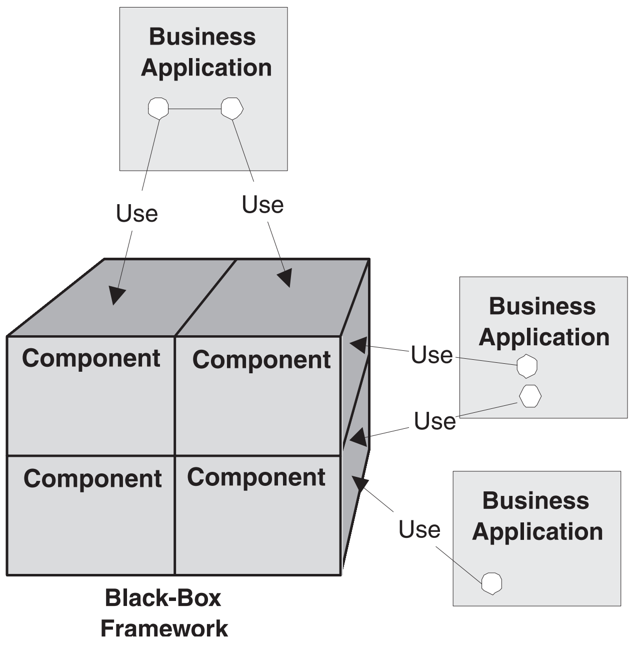

Figure 2-8. A black-box framework

Because of the composition approach in a black-box framework, it provides a greater range of flexibility than that of a white-box framework. Developers can pick and choose different components to achieve specific application requirements with infinite possibilities. Unlike white-box frameworks, where a developer often needs to know the detailed implementation of the framework component for adaptation, black-box frameworks consist of components that hide their internal implementation. Adaptation of such components is done through well-defined interfaces, such as certain public methods and properties. Developers need to be familiar with only these public members in order to use the framework.

Compared with white-box frameworks, black-box frameworks are harder to develop. Encapsulating business-domain expertise into components that are generic enough to be used in many application scenarios is not an easy task. Encapsulating too much will lead the domain expertise inside the component becoming less fit in many application scenarios. Encapsulating too little will lead to developers having to work with a large number of components and more complex coordination logic in order to build the application.

The extra flexibility of black-box frameworks doesn't come for free. When using a black-box framework, developers must implement their own process flow and coordination logic needed to link multiple components together. Because developers now control how and what components need to work together, they are responsible for the extra workload on "wiring" the components together along with the extra flexibility provided by the black-box framework. In contrast, white-box frameworks automatically handle the "wiring" for you in the template methods of their components.

Although developers don't have to deal with learning the internal implementation of the abstract class as they do with a white-box framework, they do have to be familiar with a greater number of components and their use when using a black-box framework, since the developer now has more "moving" parts to deal with in order to combine them into something they need.

When developing an application framework, there is no requirement that the framework contain either all abstract classes or all concrete classes. In fact, neither pure white-box nor black-box frameworks are practical in the real world. Having a mix of both the inheritance approach and composition approach gives you the freedom to use whatever approach is best for the design of a particular component. By mixing white-box frameworks and black-box frameworks, you effectively create a gray-box framework.

Gray-Box Frameworks

Gray-box frameworks take both inheritance and composition approach, is usually made up with combination of abstract classes and concrete classes. The approach of enabling its common spots and hot spots is determined on a component-by-component basis. Figure 2-9 shows a gray-box framework.

NOTE *Because a concrete class that contains virtual methods can either be used directly or inherited by another class that overrides the virtual methods to alter its behavior, it is possible for a gray-box framework to consist of only concrete classes, as long as the approach taken by the classes is a combination of composition and inheritance.*

In the real world, the application frameworks you will be developing will most likely be gray-box frameworks. The decision whether a given components will follow the inheritance approach or composition approach is decided by which approach is best suited for what the component is trying to accomplish and how developers will likely use the component in their business applications.

As we are choosing the inheritance approach, the composition approach, or a mix of the two, we should consistently keep in mind the tradeoffs and implications for performance, maintenance, and usability with each approach.

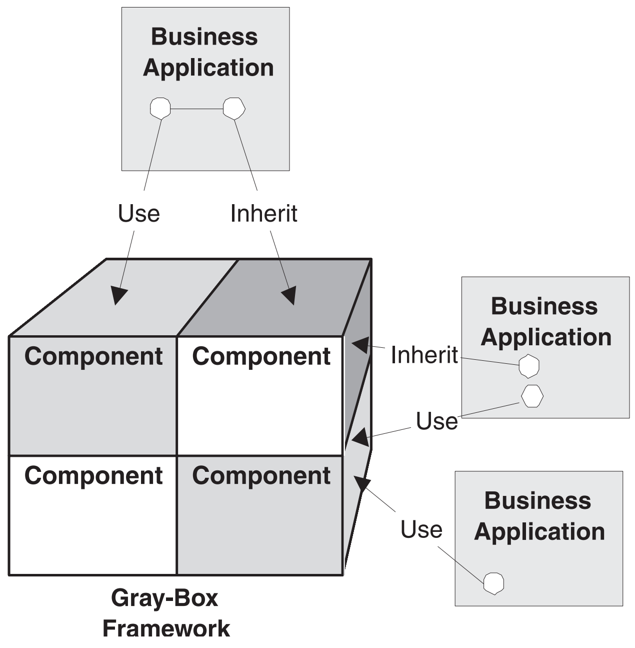

Figure 2-9. A gray-box framework

From the performance point of view, the composition approach tends to be slower than the inheritance approach, primarily because of the extra components it has to load and access at run time to produce the desired results. As you are gathering the features from multiple components, you may also add a number of extra calls to bridge different components. In the inheritance approach, however, an inherited class often contains most of the required features within itself, hence reducing the number of objects the program needs to create and access to produce the same results and eliminating as well much of the extra code that would be needed to bridge different components if the composition approach were used.

Maintenance is another area in which we see the tradeoffs in both approaches. In the composition approach, developers work with a set of highly decoupled framework components, which makes their application more adaptive to changes, and hence more flexible and extensible. However, after the application is deployed, those who provide postproduction support will have to deal with many more "moving parts," which leads to extra maintenance effort. For example, a change in a business requirement may result in the modification of framework components. With a composition approach, such requirement changes may potentially affect a series of framework components that participate in a certain business feature, since the business requirement is supported by the collaboration of a number of components. Such changes in multiple components may also multiply the effort in testing and deployment of the application framework. On the other hand, the inheritance approach may be less flexible than the composition approach, but in compensation, it introduces fewer moving parts. When business features are served by a hierarchy of classes, a change in business requirements can often be resolved with changes to far fewer classes on the hierarchical tree. Of course, the real maintenance cost of your application framework depends on the design of the framework as well as the type of business-requirement changes involved. But generally speaking, you have less overhead on maintenance if you have fewer moving parts to deal with.

Usability is yet another area you need to consider in designing the application framework. The framework component that takes on the inheritance approach usually hides the complex coordination logic and process flow inside the parent class or abstract class, so the developer often needs only to implement or override a few methods to achieve the desired result. Hence, inheritance provides very good usability as long as developers aren't required to learn overwhelming details of the parent class or abstract class they inherit. Usability for the composition approach, on the other hand, depends considerably on how much composition developers have to support to achieve certain results. Having a set of highly decoupled components often requires developers to learn and code their own coordination logic and process flow to wire such components together to produce the desired results. However, if you are willing to sacrifice flexibility and create a few coarse-grained components so that developers need to work with only a small number of components to get what they want, then the framework will become easier to use and developers will be much more productive, since the composition approach significantly reduces the coordination logic developers have to learn and write.

As you approach many framework design decisions, you must keep in mind that there is no silver bullet. There is often a tradeoff between different approaches. It is your job to decide how to balance them and create an application framework that fits your objectives.

Design Patterns

As you are architecting and developing the application framework, you will often run into design challenges on recurring scenarios, such as how to improve handling of changes to the process flow and how to improve application-specific customization. Design patterns, which describe the solution to common software development problems, can assist you in solving some of these common problems in developing an application framework. Many commonly used design patterns are documented in the classic book *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, the "gang of four" (GOF). Some design patterns are especially useful in application framework development. The following list describes some of these patterns and the problems they can solve:

Strategy: a design that handles the variation of algorithms in the application. It allows the developer to customize the framework by "plug and play"-ing different application-specific algorithms.

Bridge: a design that decouples the abstraction and implementation in the application. It allows developers to provide different implementations for part of the application without affecting other parts of the application.

Decorator: a design that provides a layer approach in processing data. It allows developers to easily assemble multiple components to process data.

Observer: a design that provides a publish–subscribe communication model. It allows developers to disperse information easily to multiple objects.

Mediator: a design that keeps objects from referring to each other explicitly. It allows developers to create loosely coupled communication between different objects.

Template method: a design that provides the skeleton of the algorithm it operates. It allows developers to define process flow and coordination logic without having to define how the algorithm is implemented.

Visitor: a design that lets you define a new operation without changing the existing ones. It allows developers to decouple an operation from the coordination logic that is constantly changing.

Singleton: a design that ensures that only one instance of the class is created. It allows developers to have better control of the creation of the object.

Abstract factory: a design that provides an interface for creating families of objects without specifying their concrete classes. It allows developers to reduce the reference to concrete classes throughout the application, and hence reduce the amount of code changed when the concrete classes change.

For the rest of the book, we will look at how these design patterns can help us develop our application framework and how these patterns are implemented in .NET.

Summary

In this chapter, you have learned about processes and techniques of application framework development. We first looked at the different layers that make up the application framework and how each layer is related to the others. Then we looked at the framework development process, which involves analysis, design, development, and stabilization stages in an iterative fashion and specific tasks involved in each of these stages. Following that, you learned about the several approaches in framework development, such as white-box, black-box, and gray-box frameworks. We also looked at some key framework development techniques through discussion of common-spot, hot-spot, and design patterns. For the rest of book, we will consider how to actually design and implement an application framework through examples. We will use a sample framework as reference to show you how you can develop your application framework using .NET technology and design patterns.