JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2006**

Vol.5, No.8, November - December 2006

Structural Design Patterns and .NET Framework 2.0

Dhamayanthi N, Senior Manager, Talent Transformation, HCL Technologies Ltd, Chennai, India
Thangavel P, Prof. & Head, Dept. of Computer Science, University of Madras, Chennai, India

There are many uncertainties in identifying and applying suitable design patterns while designing applications. Our goal is to investigate applications of design patterns in .NET Framework. Among the seven structural patterns described by Gang of Four (GOF), we have discussed five patterns from .NET perspective. As there is no official documentation available on the patterns used in the latest version of .NET Framework (version 2.0) currently, this study would be beneficial for .NET designers in understanding patterns behind the framework, which in turn would help them in applying relevant patterns while developing their aplications in .NET Framework.

1 INTRODUCTION

Design Patterns provide generic, reusable designs that solve problems at the design level. A design pattern does not define code and is not specific to a given programming domain. Instead, it provides a proven, tested solution for a class of similar design problems. Design Patterns also lend common terminology that we can use to make our own designs easier to document and understand.

Patterns describe techniques that experts have abstracted from multiple specific solutions. Identifying and understanding patterns provide two essential benefits [6]. First, they introduce effective design strategies to less experienced designers, alleviating them from rediscovering these patterns using trial and error. Second they attach common names to ideas that we can readily use in conversation, design meetings, and documentation.

Design patterns are not created from theoretical examples. Experienced object-oriented designers recognize that certain object structures and interactions lend themselves more readily to maintenance and reusability than others. Just like architects and civil engineers define broad building concepts that apply at many levels, software designers assemble a set of patterns that they can apply to a variety of domains.

Cite this article as follows: Dhamayanthi N, Thangavel P: “Structural Design Patterns and .NET Framework 2.0”, in *Journal of Object Technology*, vol. 5, no. 8, November - December 2006, pp. 59-80 http://www.jot.fm/issues/issue_2006_11/article1

Many seasoned designers have taken the initiative to identify and document patterns. Design Pattern catalogs consolidate related patterns in a single reference collection. The most influential design pattern catalog is the book Design Patterns: Elements of Reusable Object-Oriented Software by Erick Gamma, Richard Helm, Ralph Johnson, and John Vlissides [1]. It identifies and describes 23 patterns that solve broad object-oriented design problems under three categories: Creational, Structural and Behavioral.

Creational patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed and represented. A class creational pattern uses inheritance to vary the class that is instantiated. An object creational pattern will delegate instantiation to another object. Creational patterns give us a lot of flexibility in what gets created, who creates it, how it gets created and when.

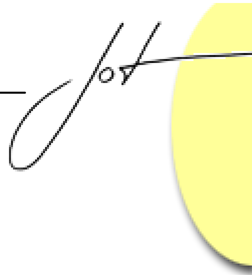
Structural Patterns are concerned with how classes and objects are composed to form larger structures. It uses inheritance to compose interfaces or implementations. Structural Patterns describe ways to compose objects to realize new functionality.

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavior Class Patterns use inheritance to distribute behavior between classes. Behavior Object Patterns use object composition than inheritance.

The .NET Framework is a suite of products for developing and managing systems with tiered and Object Oriented architectures. A mature framework usually incorporate several design patterns so is .NET framework. The patterns help make the framework's architecture suitable for many different applications without redesign. An added benefit comes when the framework is documented with the design pattern it uses. This paper is intended to present the .NET versions of the chosen structural patterns. Among the seven structural patterns discussed by Gang of four (GOF), five are explored from the .NET context. A familiarity with design patterns will assist .NET developers, regardless of the language in which they choose to develop, in using the .NET Framework.

This study is not about proposing a new pattern but about presenting the existing patterns in a language .NET designers can understand. The well-known Design Patterns text by the GOF discusses 7 Structural patterns. Among them, five patterns: Adapter, Bridge, Composite, Decorator and Proxy are discussed.

Section 2 shows the architecture of .NET Framework. Sections 3 to 7 explore the chosen five structural patterns. These sections have a brief intent followed by a discussion section in which the pattern's motivation is discussed. UML class diagram for the patterns is given to make the understanding better. Implementation section considers issues in implementing the pattern in C#. The next section presents the usage of patterns in .NET Framework and the last section talks about the relations between other structural patterns. Conclusion is given in Section 8.



2 .NET FRAMEWORK

.NET Framework is the Microsoft latest technology for developing enterprise-scale ASP.NET Web applications and high performance desktop applications. In addition Microsoft .NET Framework is a platform for building, deploying, and running Web Services and applications. It provides a highly productive, standards-based, multi-language environment for integrating existing investments with next-generation applications and services as well as the agility to solve the challenges of deployment and operation of Internet-scale applications. The .NET Framework consists of two main parts: the common language runtime (CLR) and a unified, hierarchical class library that includes a revolutionary advance to Active Server Pages called ASP.NET, an environment for building smart client applications (Windows forms) and a loosely-coupled data access subsystem ADO.NET

Figure 1: .NET Framework

Comprehensive class libraries

One of the benefits of .NET environment is their large set of base classes for all types of application development. The data types in the framework possess wide-ranging capabilities. Such broad functionality eliminates certain design considerations and even the requirements for some design pattern implementations. For instance, we need not design and implement internal Iterators in .NET because the classes under *System.Collections* namespace provide their own iteration methods.

The System.Collections.Generic namespace contains interfaces and classes that define generic collections, which allow users to create strongly, typed collections that, provide better type safety and performance than non-generic strongly typed collections.

For software development organizations, a renewed (or first) interest in design patterns is especially relevant today because the common language runtime (CLR) in .NET, within which all managed code executes, is object oriented to the core. This means languages targeted for the common language runtime such as Visual Basic .NET, C#, C++ and J# are fully object oriented and developers can use them to build applications with object-oriented techniques.

3 ADAPTER

Intent

Covert the interface of a class into another interface clients expects. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces [1].

Discussion

An adapter class implements an interface known to its clients and provides access to an instance of a class not known to its clients. Adapter pattern is basically used to convert the programming interface of one class into that of another.

There are two ways to do this: by inheritance and by object composition. In the first case, a new class is derived from the nonconforming one and the methods are added to make the new derived class match the desired interface. The other way is to include the original class inside the new one and create the methods to translate calls within the new class. These two approaches, called class adapters & object adapters, are both fairly easy to implement.

The adapter pattern allows us to fit new elements into our design without compromising it. By creating adapters, we preserve our design's integrity and don't let low-level details or a clunky interface "leak out" and affect other objects [7].

UML Diagram

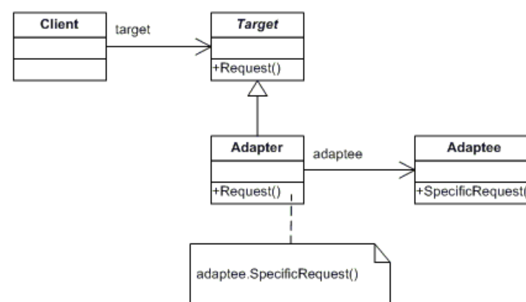
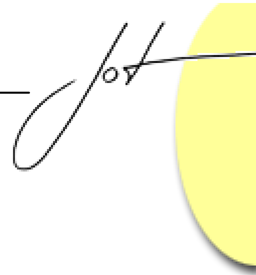


Figure 2: Adapter

Implementation Issues

The Adapter pattern frees us from worrying about the interfaces of existing classes when we are doing a design [9]. There are also some differences between the class and the object adapter approaches while implementing them using C#, although they are less significant than in C++.



Consequences in choosing Class Adapters

1. The class adapter won't work when we want to adapt a class and all of its subclasses, since a class adapter adapts Adaptee to target by committing to a concrete Adapter class.
2. Since Adapter is a subclass of Adaptee, the class adapter lets the Adapter override some of Adaptee's behavior.

Consequences in choosing Object Adapters

1. The object adapter could allow subclasses to be adapted by simply passing them in as part of a constructor. Hence this option lets a single adapter work with many adaptees.
2. Subclassing Adaptee is required when there is a necessity to override Adaptee's behavior.

Uses of Adapter in .NET Framework

Adapting data in .NET

A major goal of n-tier architecture is to define how data is represented in persistent storage, business objects and in visual presentations. Apart from this, Architecture must also supply mechanisms for transforming data between these representations, as there will be a frequent need to adapt data in one tier to meet the needs of another tier.

DataAdapter class in .NET Framework represents a set of data commands and a database connection that are used to fill the *DataSet* and update the data source. The *DataAdapter* serves as a bridge between a *DataSet* and a data source for retrieving and saving data. The *DataAdapter* provides this bridge by mapping *Fill,* which changes the data in the *DataSet* to match the data in the data source, and *update* that changes the data in the data source to match the data in the *DataSet*.

.NET framework (ver 1.1 and ver 2.0) has implementations of the following data adapters:

SqlDataAdapter

OledbDataAdapter

OdbcDataAdapter and

OracleDataAdapter

The *SqlDataAdapter* serves as a bridge between a *DataSet* and *SQL Server* for retrieving and saving data. The following code snippet creates an instance of a *DataAdapter* that uses a connection to the *SQL Server Northwind* database and populates the *DataSet* with the list of employees.

```
using (SqlConnection connection = new
SqlConnection(connectionString))
// connectionString retrieves the connection string from
// a configuration file
{
SqlDataAdapter adapter = new SqlDataAdapter("select * from
employees", connection);
DataSet ds = new DataSet();
Adapter.Fill(ds, "Employees");
}
```

In the above example we have

Adaptee	-	*Data Source (SQL Server)*
Target	-	*DataSet*
Adapter	-	*SqlDataAdapter*
Target Method	-	*Fill (Dataset instance)*

Adapters for Character Encoding and Decoding

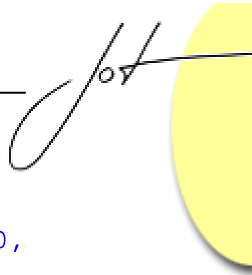
Characters are abstract entities that can be represented using many different character schemes or code pages. For example, *Unicode UTF-16* (Unicode Transformation Format, 16-bit encoding form) encoding represents characters as sequences of 16-bit integers, while *Unicode UTF-8* represents the same characters as sequences of 8-bit bytes. The common language runtime uses *Unicode UTF-16* to represent characters.

Applications that target the common language runtime use encoding to map character representations from the native character scheme to other schemes [15]. Applications use decoding to map characters from non-native schemes to the native scheme. *System.Text* namespace contains the classes such as *Encoding*, *ASCIIEncoding*, *UnicodeEncoding* and *UTF8Encoding* to encode and decode characters that act as adapter classes.

The following code example converts a Unicode string into an array of bytes using the *ASCIIEncoding GetBytes* method. Each byte in the array represents the ASCII value for the letter in that position of the string.

```
string MyString = "Encoding String";
ASCIIEncoding AE = new ASCIIEncoding();
Byte[] ByteArray = AE.GetBytes(MyString);
for (int x=0; x <= ByteArray.Length - 1; x++)
{
   Console.Write("{0} ",ByteArray[x]);
}
```

[C#]



```
Dim AE As New ASCIIEncoding()
Dim ByteArray As Byte() = { 69, 110, 99, 111, 100, 105, 110,
103, 32, 83, 116, 114, 105, 110, 103, 46 }
Dim CharArray As Char() = AE.GetChars(ByteArray)
Dim x As Integer
For x = 0 To CharArray.Length - 1
   Console.Write(CharArray(x))
Next
```

[Visual Basic]

In the above example, we have

Adaptee	-	*String*
Target	-	*Byte*
Adapter	-	*ASCIIEncoding*
Client	-	*the class that wants to convert Unicode string into Array of Bytes*
TargetMethod()	-	*GetBytes(string)*

Adapters for Interoperation with COM

An assembly that represents a COM object to a .NET client or vice versa is called an *interop* assembly. There are two kinds of *interop* assemblies, *RCW* (Runtime Callable Wrapper) and *CCW* (COM Callable Wrapper).

A .NET client accesses a COM server by means of *RCW* as shown in figure 3 below. The *RCW* wraps the COM object and mediates between it and the common language runtime environment, making the COM object appear to .NET clients just as if it were a native .NET object and making the .NET client appear to the COM object just as if it were a standard COM client. A COM client accesses a .NET object through *CCW*. The *CCW* wraps up the .NET object and mediates between it and the common language runtime environment, making the .NET object appear to COM clients just as if were a native COM object.

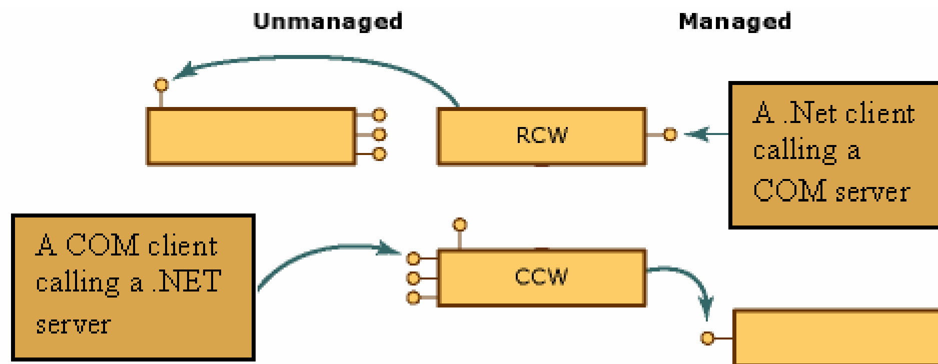


Figure 3: Interop between managed and unmanaged code

There are three ways of producing interop assemblies [14]

1. Using the tools provided by Visual Studio.NET
2. Using the type library importer tool, *TlbImp.exe*, which is part of the .NET Framework
3. Using the *System.Runtime.InteropServices.TypeLibConverter* class

If for example, COM type library includes a *dispinterface* method named *SomeMethod* that returns *void* and has an *[out, retval]* parameter as given below

```
void SomeMethod([out, retval] VARIANT_BOOL*);
```

TlbImp.exe tool produces the following method signature for *SomeMethod* in the RCW

```
void SomeMethod(out bool x);
```

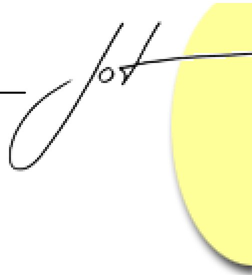
If we use */transform:dispret* option while using *TlbImp.exe*, the following method signature would be generated in the *RCW*

```
bool SomeMethod();
```

By allowing managed classes and COM components to interact despite their interface differences, interop assemblies (*RCW* and *CCW*) are examples of adapter pattern.

Delegates as an Adapter

Delegates are reference types that encapsulate methods. They are object oriented, type safe & secure. The following example declares a delegate named *Del* that encapsulate a method that takes an integer as an argument and returns void



```
public delegate void Del(int n);
```

The following two methods (static & instance methods) are created for the delegate to wrap

```
static void Delegate_Static_Method(int i)
{
   Console.WriteLine("value passed to the static method is
{0}",i);
}
void Delegate_Instance_Method(int j)
{
   Console.WriteLine("value passed to the instance method is
{0}",j);
}
```

Delegate objects *d1* and *d2* are instantiated by assigning the names of the methods *Delegate_Static_Method* and *Delegate_Instance_Method* respectively.

```
Del d1 = Delegate_Static_Method;
Del d2 = obj.Delegate_Instance_Method;
// obj is an object of the class which contains
// Delegate_Instance_Method
```

The instantiated delegate objects *d1* and *d2* can be invoked as if they were the wrapped methods. The parameters passed to the delegates by the caller are passed to the methods, and the return value, if any, from the methods are returned to the caller by the delegates.

```
d1(5);
d2(10);
```

When we define a delegate, the compiler creates a sealed class deriving from CLR-provided *MulticastDelegate* class which in turn is derived from an abstract class called *Delegate*. *Delegate* class has two public properties *Target* and *Method*. Target gets the class instance on which the current delegate invokes the instance method. Method gets the method represented by the delegate.

The delegate is an example of Adapter pattern. The purpose of an adapter is to convert class's interface into one that client's expect. A delegate is an adapter because it converts the interface of a function pointer to another interface. Instead of working with a function pointer, clients work with the new interface exposed by the delegates.

Related structural patterns
Bridge, Façade and Decorator

An Adapter changes the interface of the Adaptee, at least from the client's perspective. Other patterns are also used to present a different messaging interface to a client. In the Bridge pattern, the Abstraction object defines an abstraction of its implementation object's interface. The goal is to hide implementation details behind the Abstraction interface. The Abstraction object effectively translates messages and forwards them to the implementation object, where the real work is performed. Bridge has a different intent and applicability from Adapter however.

Adapter is applicable when we have objects that only know how to communicate with their collaborators using a certain protocol but need to collaborate with objects that don't understand this protocol. Bridge is not meant to adapt an existing object to an incompatible protocol.

A Façade provides an abstract interface to an entire subsystem of objects, whereas an Adapter adapts a single object to its client. Although their implementations and applicability are quite different, the two patterns are similar in spirit, in providing an alternative interface for clients.

A Decorator also sits between a Client and another object, receiving client messages and forwarding some or all to the Component. A decorator, though, completely adheres to the interface of its component is used to enhance its functionality. Unlike an Adapter, it does not adapt the interface of the component it decorates.

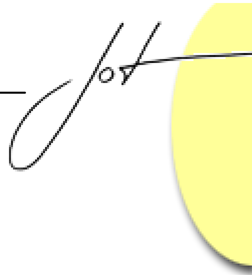
4 BRIDGE

Intent

Decouple an abstraction from its implementation so that the two can vary independently [1].

Discussion

The Bridge pattern is useful when there is a hierarchy of abstractions and a corresponding hierarchy of implementations. Rather than combining the abstractions and implementations into many distinct classes, the Bridge pattern implements the abstractions and implementations as independent classes that can be combined dynamically.



UML Diagram

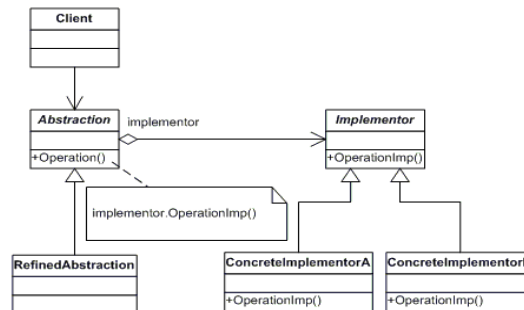


Figure 4: Bridge

Implementation Issues

1. The Bridge pattern is intended to keep the interface to our client program constant while allowing us to change the actual kind of class we display or use. This can prevent us from recompiling a complicated set of user interface modules.
2. We can extend the implementation class and the bridge class separately and usually without much interaction with each other.
3. The implementation details can be hidden from the client program much more easily.

Uses of Bridge in .NET Framework
Windows forms as Bridges

The .NET visual control is an example of a Bridge pattern implementation [5]. A control is a reusable software component that can be manipulated visually in a builder tool.

All of the C# controls support a query interface that enables builder programs to enumerate their properties and display them for easy modification. *Visual Studio.NET* displays a builder panel that is used to modify the properties of all the controls that are displayed in the form. All windows form controls have the same interface used by the Builder program and we can substitute any control for any other and still manipulate its properties using the same convenient interface. The actual program which we construct uses these classes in a conventional way, each having its own rather different methods, but from the builder's point of view, they all appear to be the same.

Database Drivers as Bridges

A driver is an object that operates a computer system or an external device according to a well-specified interface. Drivers provide the most common example of the Bridge pattern.

Applications that use drivers are abstractions. The effect of running the application depends on which driver is in place. Each driver is an instance of the Adapter pattern,

providing the interface a client expects using the services of a class with a different interface. An overall design that uses drivers is an instance of Bridge [4]. The design separates application development from the development of the drivers that implement the abstract operations on which the applications rely.

A driver-based design forces us to create a common, abstract model of the machine or system that will be driven. This has the advantage of letting code on the abstraction side apply to any of the drivers that it might execute against. Defining a common set of methods for the drivers may also incur the disadvantage of eliminating behavior that one driver entity might support.

These tradeoffs in having a broad or narrow interface in an implementation of Bridge also appear in database drivers. For eg., in .NET, we can use the *OledbDataReader* class to work with almost any database, including SQL Server. However, .NET also provides *SqlDataReader* that works only with SQL server and is faster than the *OleDbDataReader*. In addition, the *SqlDataReader* class offers methods that the *OleDbDataReader* class does not, such as the *GetSqlMoney()* method.

Adaptive Rendering Functionality – ASP.NET Server Controls as Bridges

In the .NET Framework Version 2.0, *ASP.NET* provides a unified control architecture that enables a single set of server controls to work with many different browsers for desktop and mobile devices. This functionality is made possible by pluggable adapter classes, which render markup based on the capabilities of the browser that accesses a page.

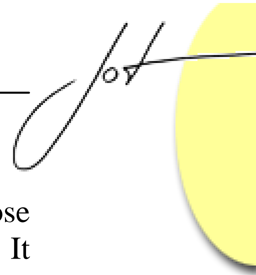
In the earlier versions of .NET Framework, *ASP.NET* mobile Web Forms pages were created using mobile controls. These controls implemented rendering for mobile-device browsers and were distinct from the set of standard *ASP.NET* server controls. In the new *ASP.NET* adaptive rendering architecture [16], a separate set of mobile controls does not exist. *ASP.NET* server controls encapsulate core user interface logic and provide default rendering, which is *HTML* for desktop browsers. Rendering for other browsers is provided by separate adapter classes, which are not controls. Adapters bind to controls at run time and adapt the output of controls to the requesting browser. Adapters call markup writer classes for creating *WML*, *XHTML*, *cHTML*, and other kinds of markup. Mappings between *controls* and *adapters* are specified in configuration files.

An important feature of adapters is that they operate behind the scenes without any work on the part of the page developer. A page developer can use a control without any knowledge of adapters, assuming that the control itself adapts its rendering for different browsers. The *ASP.NET* page framework enables page developers to use filters to customize control properties for different browsers and devices.

Related Structural patterns

Bridge vs. Decorator

Bridge and Decorator attack the same problem when we have to implement the same subclass of two different classes by not choosing the subclassing as the solution.



Decorator solves it by extracting the subclass's extra behavior into a separate class whose instances can be wrapped around the base classes' instances to add the extra behavior. It requires that all of the classes have the same interface. It also enables multiple decorators to be nested inside one another on top of a single concrete component.

Bridge extracts all of the implementation details out of the original classes and moves them into a separate implementation hierarchy. Then it reimplements the original hierarchy to make it a set of abstractions that represents subtypes and delegate their implementation to the implementors.

Abstraction subclasses can have extended interfaces, but they cannot be nested like decorators. However the Bridge pattern can be applied repeatedly so that the implementor hierarchy for one example might turn out to be the abstraction hierarchy for a second example.

Bridge and Proxy

Proxy can be implemented as a Bridge with Proxy being the abstraction and object being the implementor [8].

Adapter and Bridge

Bridge may use Adapter in its own solution. An example of this is the data structure set: Adapter can be used to view lists, arrays, and tables as sets. Thus Adapter standardizes the interfaces of the different ConcreteImplementor interface in the Bridge pattern [12].

5 COMPOSITE

Intent

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly [1].

Discussion

The composite object allows us to build complex objects by recursively composing similar objects in a tree-like manner. The composite pattern also allows the objects in the tree to be manipulated in a consistent manner, by requiring all of the objects in the tree to have common interface or superclass. During the design process the composite pattern is often used to recursively decompose a complex object into simpler objects [3].

The Composite pattern addresses situations where primitive objects can be grouped into composite objects, and the composites themselves are considered primitive objects [2].

Composite's downside is that it can lead to a system in which the class of every object looks like the class of every other [10].

UML Diagram

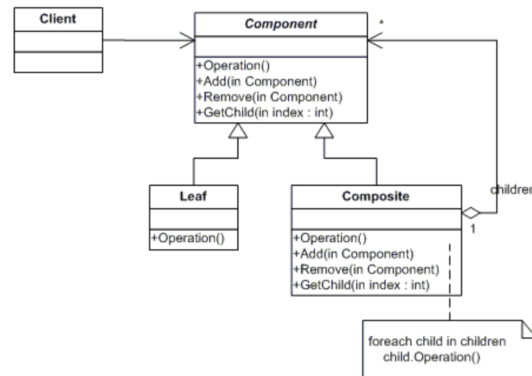


Figure 5: Composite

Implementation Issues

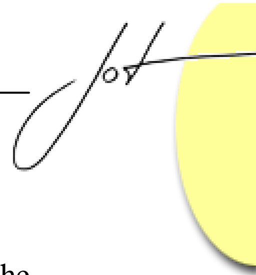
Composite patterns require new presentation and modeling techniques because their complexity makes them more difficult to approach than noncomposite patterns [13]. Composite contains two powerful, related concepts. One concept is that a group can contain either individual items or other groups. Related to this concept is the idea that groups and individual items may share a common interface. These ideas come together in object modeling, where we can create an abstract class or a C# interface that defines the behaviors that are common to groups and to individual objects.

The ability to distinguish between nodes and leaves when we have a single interface to access all the objects in a Composite is really challenging.

Some authors have suggested creating a separate interface for nodes and leaves. This then leaves us with the programming problem of deciding which elements will be which when we construct the composite. However Design Patterns by GOF suggests that each element should have the same interface whether it is a composite or a primitive element. This is easier to accomplish, but we are left with the question of what the getChild operation could accomplish when the object is actually a leaf.

C# makes this quite easy since every node or leaf can return an *ArrayList* of the child nodes. If there are no children, the *count* property returns zero. Thus, if we simply obtain the *ArrayList* of child nodes from each element, we can quickly determine whether it has any children by checking the *count* property.

A nonleaf node can have child-leaves added to it, but a leaf node cannot. However we would like all of the components in the composite to have the same interface. We must prevent attempts to add children to a leaf node, and we can design the leaf node class to raise an error if the program attempts to add to such a node.



Ordering Components

If the order of the components is different from the order in which they were added to the parent, then the parent must do additional work to return them in the correct order.

Caching results

If the data that is required frequently must be computed from a series of child components, it may be advantages to cache these computed results in the parent. Caching is recommended if the computation is relatively intensive.

Consequences

The Composite pattern allows us to define a class hierarchy of simple objects and more complex composite objects so that they appear to be the same to the client program. Because of this simplicity, the client can be that much simpler, since nodes and leaves are handled in the same way.

The composite pattern also makes it easy for us to add new kinds of components to our collection as long as they support a similar programming interface. On the other hand, this has the disadvantage of making our system overly general. We might find it harder to restrict certain classes where this would normally be desirable.

Uses of Composite in .NET Framework
TreeNodeCollection class & *ToolStrip* class in .NET Framework 2.0

The *TreeNodeCollection* class that represents a collection of *TreeNode* objects in the *TreeView* control in .NET Framework 2.0 is an example of a Composite pattern. We can also find that the Composite describes the hierarchy of Form, Frame and Controls in any user interface programs.

ToolStrip class in .NET Framework 2.0 provides container for Windows toolbar objects. *ToolStrip* is the container for *ToolStripButton*, *ToolStripComboBox*, *ToolStripSplitButton*, *ToolStripLabel*, *ToolStripSeparator*, *ToolStripDropDownButton*, *ToolStripProgressBar* and *ToolStripTextBox* objects.

Any container may then contain components such as *buttons*, *check boxes*, and *TextBoxes*, each of which is a leaf node that cannot have further children. They may also contain *ListBoxes* and *grids* that may be treated as leaf nodes or that may contain further graphical components. Using the *Controls* collection, the Composite tree can be walked down.

CompositeControl class in .NET Framework 2.0

CompositeControl class is an abstract class that implements the basic functionality required by web controls that contain child controls. We can create custom composite controls by deriving from *CompositeControl* class. *Login* control which is derived from

CompositeControl is a composite control that provides all the common UI elements needed to authenticate a user on a web site.

Related Structural Patterns

Composite and Decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects [1].

Flyweight is often combined with Composite to implement shared leaf nodes [1].

6 DECORATORS

Intent

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality [1].

Discussion

The Decorator pattern provides us with a way to modify the behavior of individual objects without having to create a new derived class. This is one of the cases where object containment is favored over object inheritance.

UML Diagram

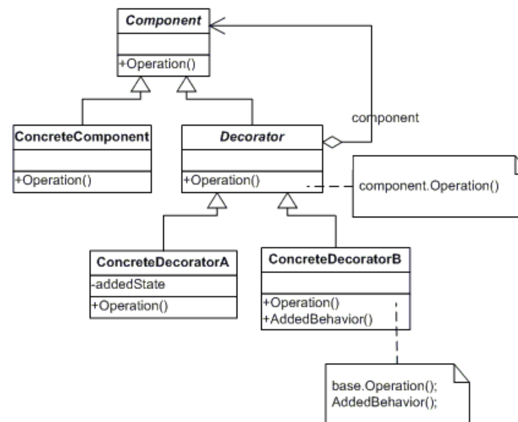
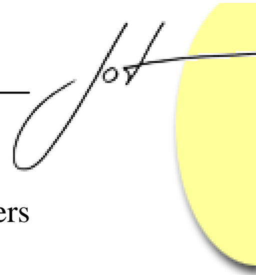


Figure 6: Decorator

Implementation Issues

Design Patterns by GOF suggests that Decorators should be derived from some general visual component class, and then every message for the actual button should be



forwarded from the Decorator. This is not all that practical in C#, but if we use containers as Decorators, all of the events are forwarded to the control being contained.

Nonvisual Decorators

Decorators are not limited to objects that enhance visual classes. We can add or modify the methods of any object in a similar fashion. Non-visual objects are generally easier to decorate because there may be fewer methods to intercept and forward. Whenever we put an instance of a class inside another class and have the outer class operate on it, we are essentially decorating that inner class. This is one of the most common tools for programming available in Visual Studio.NET

Consequences

The Decorator pattern provides a more flexible way to add responsibilities to a class than by using inheritance, since it can add these responsibilities to selected instances of the class. It also allows us to customize the class without creating subclasses high in the inheritance hierarchy.

Design Patterns by GOF [1] points out two disadvantages of the Decorator Pattern.

1. The Decorator and its enclosed component are not identical. Thus, tests for object types will fail.
2. Decorators can lead to a system with "lots of little objects" that all look alike to the programmer trying to maintain the code. This can be a maintenance headache.

Uses of Decorator in .NET Framework
Streams

The .NET Framework provides a classic example of the Decorator pattern in the overall design of input and output streams. A stream is an abstraction of a sequence of bytes, such as a file, an input/output device, an inter-process communication pipe, or a TCP/IP socket. *Stream* is the abstract base class of all streams in .NET Framework. Two of its derived classes *BufferedStream* which adds a buffering layer to read and write operations on another stream and *CryptoStream* which links data streams to cryptographic transformations have constructors that accept another stream, so that we create a stream from a stream. This sort of slim composition is the typical structure of Decorator. The Decorator pattern is at work in .NET streams, to a degree. But, with a small amount of code, we can leverage Decorator to greatly expand our ability to mix in variations of the read and write operations of streams.

Decorators in GUI

One of the original applications of Decorator lies in the construction of GUI components. Many GUI controls in .NET can contain other controls.

Related Structural patterns
Decorators, Adapters and Composites

Adapters also seem to "decorate" an existing class. However their function is to change the interface of one or more classes to one that is more convenient for a particular program. Decorators add methods to particular instances of classes rather than to all of them. We could also imagine that a composite consisting of a single item is essentially a Decorator. Once again, the intent is different.

Decorators and Façade evoke similar images in building architecture, but in design pattern terminology, the Façade is a way of hiding a complex system inside a simpler interface, whereas Decorator adds function by wrapping a class.

7 PROXY

Intent

Provide a surrogate or placeholder for another object to control access to it [1].

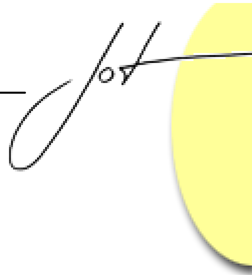
Discussion

The Proxy pattern is used when we need to represent an object that is complex or time consuming to create with a simpler one. If creating an object is expensive in time or computer resources, Proxy allows us to postpone this creation until we need the actual object. A proxy usually has the same methods as the object it represents and once the object is loaded, it passes on the method calls from the Proxy to the actual object.

There are several scenarios where a Proxy can be useful.

1. An object, such as large image, takes a long time to load
2. The results of the computation take a long time to complete, and we need to display intermediate results while the computation continues.
3. The object is on a remote machine, and loading it over the network may be slow, especially during peak network load periods.
4. The object has limited access rights, and the proxy can validate the access permissions for that user.

An image proxy can note the image and begin loading it in the background while drawing a simple rectangle or other symbol to represent the image's extent on the screen before it appears. The proxy can even delay, loading the image until it receives a paint request and only then begin the process.



UML Diagram

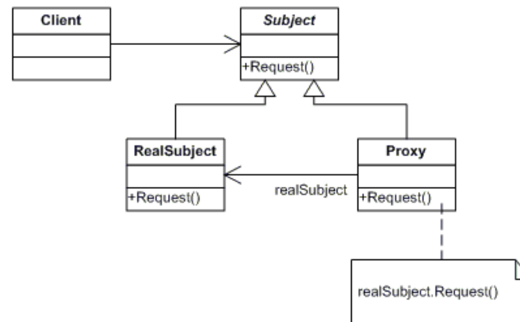


Figure 7: Proxy

Implementation Issues

A Proxy object usually has an interface that is nearly identical to the interface of the object for which it is a proxy or substitute. The proxy does its work by judiciously forwarding requests to the underlying object that the proxy controls access to.

When the proxy pattern appears in a design, its presence should be justified because the use of forwarding can create problems that other designs may avoid.

Frank Buschmann et al. [11] describe seven variants of the generic proxy pattern.

1. Remote Proxy: Clients of remote components should be shielded from network addresses and inter-process communication protocols.
2. Protection Proxy: Components must be protected from unauthorized access.
3. Cache Proxy: Multiple local clients can share results from remote components.
4. Synchronization Proxy: Multiple simultaneous accesses to a component must be synchronized.
5. Counting Proxy: Accidental deletion of components must be prevented or usage statistics collected.
6. Virtual Proxy: Processing or loading a component is costly, while partial information about the component may be sufficient.
7. Firewall Proxy: Local clients should be protected from the outside world.

Uses of Proxy in .NET Framework
A Data Proxy

Applying the Proxy pattern can create a maintenance burden, so when we use Proxy, the context of the problem that we are solving should justify our design. One such context occurs when a subsystem that we do not control supplies us with an object over which we would like to have more control. For example, the .NET *OleDbDataReader*, *OdbcDataReader* and *OracleDataReader* classes provide "data reader" objects that let us

read data from a database table. These classes are sealed so we cannot subclass them. If we want to control the behavior of a data reader object, Proxy provides an answer [4].

OldDbDataReader, *OdbcDataReader* and *OracleDataReader* implement an *IdataReader* interface. We can create a new class that implements this interface. An instance of our class can selectively forward calls to an underlying instance of *OleDbDataReader*, *OdbcDataReader* and *OracleDataReader*.

The Proxy pattern's reliance on forwarding usually creates a maintenance burden. For example, when the .NET readers change, we need to update the data reader proxy. To avoid this burden, we should usually consider alternatives to proxy, but there will be times when Proxy is the right choice. In particular, when the object for which we need to intercept messages is executing on another machine, there may be no substitute for Proxy.

Remote Proxies (ASP.NET Web Service and .NET Remoting)

We should be able to call methods on a proxy object that forwards the calls to the real object on the remote machine. *ASP.NET* makes it possible for a client to obtain a proxy object that forwards calls to a desired object that is active on another computer. *ASP.NET* Web Service is a very good example of remote proxies.

A central benefit of *ASP.NET* is that it lets client programs interact with a local object that is a proxy for a remote object. The interface for the proxy will match the interface of the remote object so client developers hardly need to be aware of the remote calls that take place. *ASP.NET* supplies the communication mechanism and isolates both server and client from the knowledge that two implementations of Web Service are collaborating to provide nearly seamless interprocess communication.

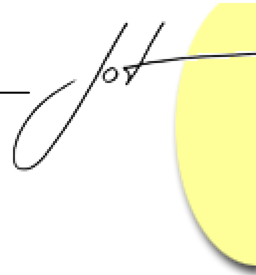
.NET Remoting enables us to build widely distributed applications easily, whether the application components are all on one computer or spread out across the entire world. We can build client applications that use objects in other processes on the same computer or any other computer that is reachable over its network. We can also use *.NET Remoting* to communicate with other application domains in the same process [16].

In the general sense, a proxy creates the illusion that a remote object is actually in the same process as the client. It does that by implementing the same methods and properties as the remote object. This is true for *.NET Remoting* proxies as well [15].

Related Structural patterns

Both Adapter and the Proxy continue a thin layer around an object. However the adapter provides a different interface for an object, whereas the Proxy provides the same interface for the object but interposes itself where it can postpone processing or data transmission effort.

A decorator also has the same interface as the object it surrounds, but its purpose is to add additional function to the original object. A Proxy, by contrast, controls access to the contained class.



8 CONCLUSION

A good working knowledge of design patterns would help the developers and designers to design more robust solutions. We have discussed the usage of five structural patterns (Adapter, Bridge, Composite, Decorator and Proxy) in .NET Framework 2.0 and the issues specific to C# implementations.

Identifying patterns in frameworks provides values in several ways. First of all, it can help people understand how the framework's feature works to extract the framework's specific context. Second, people often move between programming environments, so they might be familiar with a particular solution but not how to implement it in a new environment. Understanding the underlying pattern behind framework features helps a great deal in making this connection. Finally, even if a framework implements a pattern, we must still decide how to use it. Understanding patterns in the framework would help us in knowing more about what implementation strategies the framework uses and whether they are appropriate for a particular problem. Good understanding of development frameworks can increase the productivity of software development in any organization.

REFERENCES

[1] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[2] Cay Horstmann. *Object Oriented Design & Patterns.* John Wiley, 2004.

[3] Mark Grand. *Patterns in Java-Volume 1.* Wiley Publishing, 2002.

[4] Steven John Metsker. *Design Patterns in C#.* Pearson Education, July 2004.

[5] James W. Cooper. *C# Design Patterns A Tutorial.* Pearson Education, 2003.

[6] Clifton Nock. *Data Access Patterns.* Pearson Education, 2004.

[7] Rebecca Wirfs-Brock and Alan McKean. *Object Design Roles, Responsibilities and Collaborations.* Pearson Education, 2003.

[8] Sherman R. Alpert, Kyle Brown and Bobby Woolf. *The Design Patterns Smalltalk Companion.* Addison-Wesley, 1998.

[9] Alan Shalloway and James R. Trott. Design Patterns Explained A new perspective on Object Oriented Design. Addison Wesley, 2002.

[10] John Vlissiddes. *Pattern Hatching Design Patterns Applied.* Addison Wesley, 1998.

[11] Buschmann, F, Meunier R, Rohnert H, Sommerlad P and Stal M. *Pattern-Oriented Software Architecture – A System of Patterns.* Wiley, 1996.

[12] James O. Coplien and Douglas C Schmidt. *Pattern Languages of Program Design.* Addison-Wesley, 1995.

[13] Martin R, Riehle D and Buschmann F. *Pattern Languages of Program Design 3*. Addison-Wesley, 1998.

[14] Julian Templeman and John Paul Mueller. *COM Programming with Microsoft.NET.* Microsoft Press, 2003.

[15] Tom Barnoby. *Dsitributed .NET Programming in C#*. Apress, 2002.

[16] MSDN Library, Visual Studio 2005

About the authors

Dhamayanthi N. is a Senior Manager in the Talent Transformation team of HCL Technologies Ltd. and heads the Technical Training function. She is a Gold Medalist in MCA and currently pursuing her Ph.D. in Computer Science in University of Madras. Apart from offering design consultancy for .NET projects, she designs & delivers highly customized training programs in .NET.

She manages a sub chapter of Chennai .NET User Group which concentrates on .NET Design, Architecture & Project Management. The group has 600+ members who are designing solutions in .NET in various companies. She regularly offers guest lectures in all .NET forums across India. She has been awarded MVP (Most Valuable Professional) by Microsoft for C#.NET. She is also Microsoft Certified Trainer (MCT). She can be reached at dhamayanthin@hotmail.com.

Thangavel P. received the M.Sc. degree in Mathematics from Bharathiar University, in 1985, M.Tech. degree in Computer Science and Data Processing from Indian Institute of Technology, Kharagpur in 1988 and the Ph.D. degree in Computer Science from Bhrarathidasan University in 1995. From 1988 to 1995 he worked as a Lecturer in Mathematics at Bharathidasan University. From 1995 to 2003 he worked as Reader in Computer Science at University of Madras. Since 2003 he has been working as a Professor in Computer Science at University of Madras. His research interests include Parallel and Distributed Algorithms and Dynamic Chaotic Neural Networks. He can be reached at thangavelp@yahoo.com