Design Patterns

by Christopher G. Lasater

Chapter 4: Structural Patterns

Overview

Structural patterns are patterns whose sole purpose is to facilitate the work of changing the structural associations of classes, class associations, and hierarchies of class structures. Most structural patterns take on the form of facades or types of proxies and adapters. There is one exception to this — the Flyweight pattern. This pattern is also a structural pattern in that it provides an adaptation of sorts, albeit a disjointed one. Since it does have the definition of changing the structural association of classes in the form of shared instances as opposed to individual ones, it is included with this group of patterns. We will discuss the Flyweight pattern later in the chapter. For now we will concentrate on patterns like the Adapter pattern, which is a more solid example of a structural pattern.

Adapter Pattern

What Is an Adapter Pattern?

The *Adapter* pattern is a useful pattern for making two different class types communicate in a similar manner. It performs much as its name implies: it creates an adaptation between two classes of different types so they can become interchangeable.

The Adapter pattern has three important components: the *Target*, *Adapter*, and *Adaptee*. The target is the class for which we wish to implement the adapter. We inherit from the target to create our adapter. The adapter is the class that provides the join for the two disparate class types and houses the methods for conjoining functionality between them. It is an inherited member from the target. The adaptee is the class that we wish to give access to the methods and functionality of the target. The adapter allows interchangeability between the target and the adaptee.

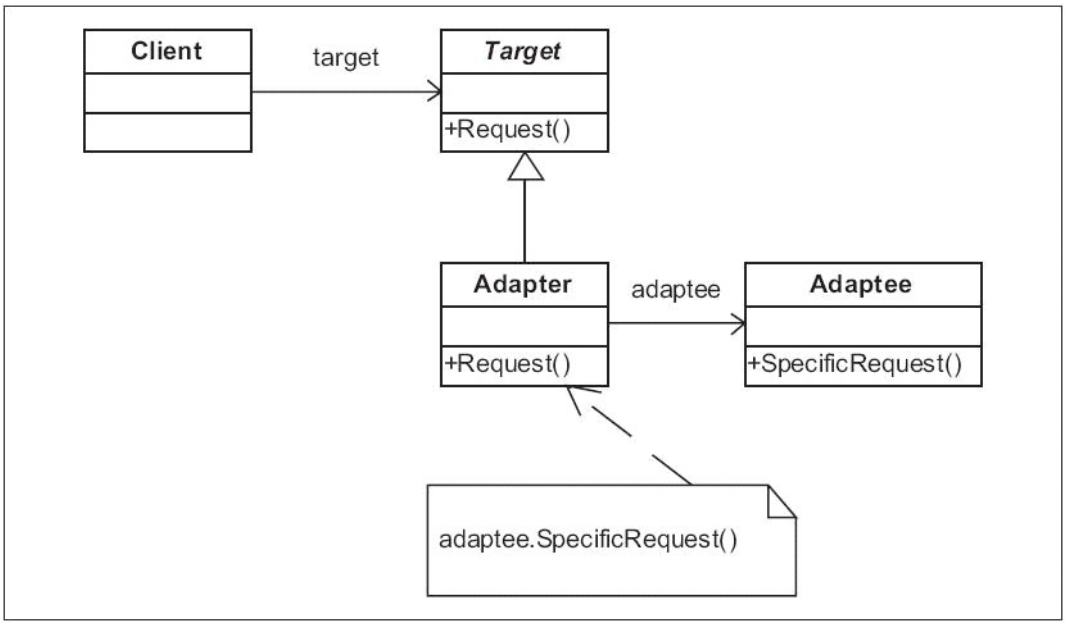

Figure 4-1: UML for Adapter pattern

This seems to be a simple enough pattern. It can also be quite useful when dealing with incompatible class types in a class solution where it is desired that neither class have any real association directly with the other, but that they use similar methods. In other words, the adapter allows the target to perform normally on the outside but inside it will actually use the adaptee's methods and functionality.

Problem: We wish to allow one class to use another's methods without using direct inheritance

For our functional problem I have chosen a simple equation. I have a class named `Water`, which houses some basic functions to manipulate temperature limits for different levels of heat:

```
//Adaptee
```

```
class Water
{
    public void SetBoilingTemperature(float temperature)
    {
        .....

    }
    public void SetIceTemperature(float temperature)
    {
        .....

    }
    public void SetLiquidTemperature(float temperature)
    {
        .....

    }
}
```

Notice each method takes a `float` that signifies temperature. Our methods could be anything; the important detail is that we wish to share these methods with other class types without modifying the class itself or the classes with which we wish to share functionality. Let's say for this example we have a class named `Element`, which would also like to take advantage of the `Water` class's methods:

```
class Element
{
.....

    public virtual void SetVaporTemperature(float temperature)
    {
        .....
    }
    public virtual void SetFreezeTemperature(float temperature)
    {
        .....
    }
    public virtual void SetNormalTemperature(float temperature)
    {
        .....
    }
}
```

We don't want to use inheritance between each class for some reason, probably to enforce the domain-specific rules of the `Element` class, the `Water` class, or both. Yet sometimes we wish to let `Element` act as though it were `Water`, although it does not directly access `Water` through inheritance or direct association. We can see that `Element` and `Water` both have similar methods, but that they are basically different:

```
Element silicon = new Element("Silicon");
silicon.SetFreezeTemperature(-20);
silicon.SetVaporTemperature(3000);
silicon.SetNormalTemperature(105);

Water water = new Water();
water.SetIceTemperature(20);
water.SetLiquidTemperature(75);
water.SetBoilingTemperature(140);
```

To use them interchangeably at this point we would have to change one or both classes, or provide some Boolean method such as:

```
if(some condition)
{
    Element silicon = new Element("Silicon");
    silicon.SetFreezeTemperature(-20);
    silicon.SetVaporTemperature(3000);
    silicon.SetNormalTemperature(105);
}
else (some other condition)
{
```

```
    Water water = new Water();
    water.SetIceTemperature(20);
    water.SetLiquidTemperature(75);
    water.SetBoilingTemperature(140);
}
```

For many reasons we might not want to do this. Probably the most notable is the lack of any polymorphic aspect between the two classes.

Solution: Allow an adapter class to hold an instance of the desired class and adapt its methods, properties, and events through the adapter's methods, properties, and events

The solution we will use for this particular problem is to create an adapter class that can allow the `Element` to act using the `Water` class's methods. Our adapter will inherit from `Element` and will encapsulate `Water` as a private instance variable.

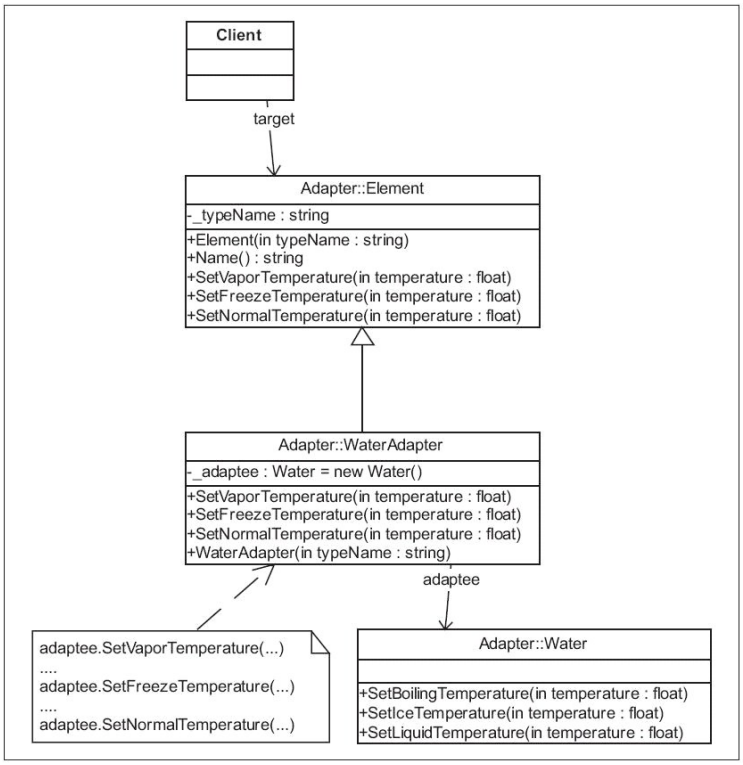

Figure 4-2: UML for Adapter pattern example

Doing this allows the adapter, which we will call the `WaterAdapter` class, to house and connect the `Water` class's methods directly into the overridden methods of the `Element` class. So now when we instantiate the `Element` class as an instance of `WaterAdapter`, we are actually calling the encapsulated `Water` class's methods as the adaptee:

```
//Adapter
class WaterAdapter : Element
{
    private Water _adaptee = new Water();

    public WaterAdapter(string typeName) : base (typeName){}

    public override void SetVaporTemperature(float temperature)
    {
        _adaptee.SetBoilingTemperature(temperature);
    }
    public override void SetFreezeTemperature(float temperature)
    {
        _adaptee.SetIceTemperature(temperature);
    }
    public override void SetNormalTemperature(float temperature)
    {
```

```
        _adaptee.SetLiquidTemperature(temperature);
    }
}
```

Now we have the option to call the `Element` class as before, but with one difference. We can now call the `Element` class as the `WaterAdapter`, and use the polymorphic qualities of the two classes to determine whether we wish to use the base element's functions of its adaptee class `Water` functions:

```
Element silicon = new Element("Silicon");
silicon.SetFreezeTemperature(-20);
silicon.SetVaporTemperature(3000);
silicon.SetNormalTemperature(105);

Element water = new WaterAdapter("H2O");
water.SetFreezeTemperature(20);
water.SetVaporTemperature(140);
water.SetNormalTemperature(75);
```

Comparison to Similar Patterns

The Adapter pattern has similarities to some other structural patterns like Bridge and Proxy, but also has some similarities to the Visitor pattern. Its similarities to the first two patterns may be obvious in that it acts as an intermediary between two class or domain areas that are themselves immutable. Its similarities to the latter pattern are in its ability to exchange functionality depending on the adaptee that is implemented, much like the Visitor pattern exchanges functionality between classes based on type.

What We Have Learned

The Adapter pattern gives us a way to provide a more common connection between uncommon class structures. If you have one class that is dissimilar to other classes you wish to emulate, you can use an adapter class to make the foreign class appear and function as the accepted type, at least in some ways. By adapting the methods, properties, or events of a class through the adapter, you can make inexact class structures work as if they were inherited from the same base. This is especially important when you wish to have different classes appear to be inheriting from the same base, without rewriting either the class to be adapted or the class for which you wish to adapt.

Related Patterns

- Bridge pattern
- Decorator pattern
- Facade pattern
- Proxy pattern
- Template pattern
- Visitor pattern

Bridge Pattern

What Is a Bridge Pattern?

Another instance of where simple inheritance cannot meet the immediate needs of the programmer is one where abstraction is not desired. When you use abstraction or inheritance, you are tied to the exact definition of that abstraction. Sometimes you would like this to be more flexible. Some cases would require classes to not be inherited, but instead we would like to adapt other classes to act as the desired type without modifying either class.

Usually an Adapter pattern would suffice to join different class types when inheritance is not desired. However, in some cases the actual implementation of the adapter needs to be more flexible. It is then that the adapter's cousin, the *Bridge* pattern, comes into play. As we saw in the "Adapter Pattern" section, the adapter houses an instance variable of the desired type to adapt as an intrinsic variable, or a private instance variable. This instance variable is not changeable in the class. This means it is not set as an abstract or base variable but as a concrete type. We hide this instance variable's methods, properties, and events behind overridden methods, properties, and events matching the adapter's base, thus

making it compatible with the expected class type the adapter is inheriting.

In the Bridge pattern we expand on this, allowing the instance variable to be not a concrete type but an abstract type, thus giving us a variance on which class we wish to adapt, or bridge. We still inherit the bridge from the expected base type and override the methods, properties, and events of that type. The difference here is that we are dependent on which concrete implementation of the abstract instance inside the bridge we have selected. Let's talk about the actual parts of the pattern and then we can find out more about this in detail.

The Bridge pattern has four main parts. The first is the *Abstraction*, which is the base for the bridge that holds the abstracted instance variable for the class to be adapted. The next part is the *Refined Abstraction*, which is the concrete implementation of the abstracted bridge. This part of the pattern is where we define the methods, properties, and events for which we wish to provide a bridge. The other half of the pattern is the *Abstract Implementor* and the *Concrete Implementor*. These two classes define the abstraction and concrete interface for the class for which we wish to bridge into the new type. The abstract implementor is the abstract instance variable to be set as the concrete implementor type at run time.

So in short, we provide a way to encapsulate a desired implementation of a class from which we do not wish to inherit. We provide methods, properties, and events in the concrete bridge based on another base type. We encapsulate the methods, properties, and events of the desired implementation class inside the bridged methods, properties, and events. The instance of the internal abstract implementation is interchangeable because of its abstract status. This allows not only polymorphism of the bridge, but also of the desired encapsulated instance implementation class inside the bridge.

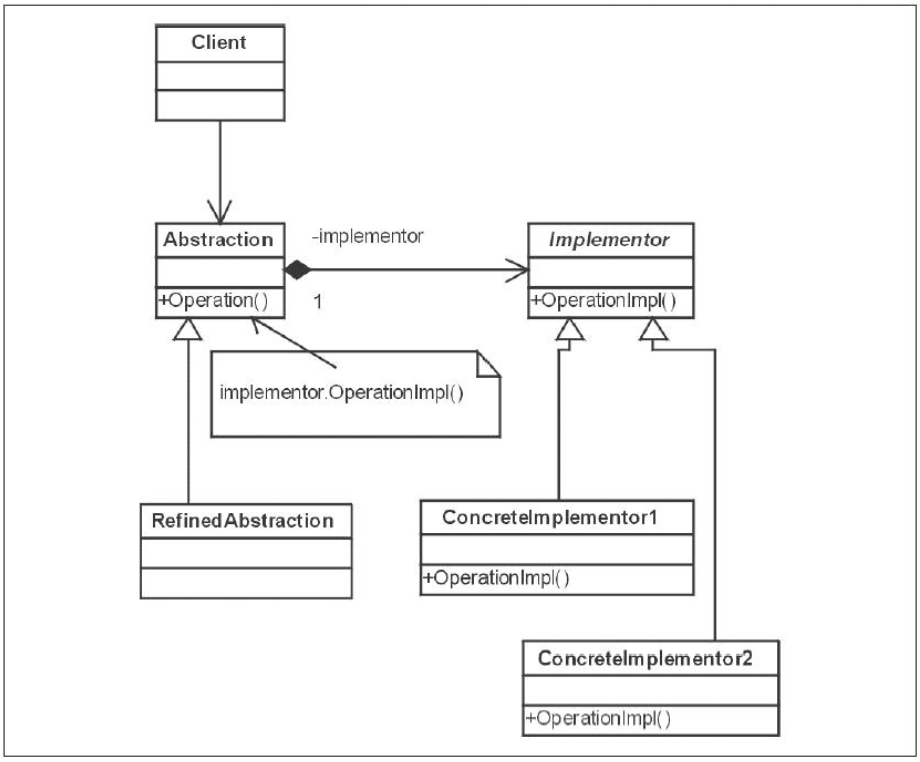

Figure 4-3: UML for Bridge pattern

Problem: We need to allow a class to use functionality from a series of class types without using direct inheritance

For our functional problem, we have a class that has a single method and inherits a base. This is a good starting point for making a bridge, because we can easily refactor the base class to be our bridge's abstraction to hold our implementor. We have an instance where our implementor is another class type that we do not want to inherit, probably because the methods do not match exactly. This is usually the case when using a bridge. You have two class types that have similar functionality, but the method, property, or event names, input parameters, etc., are slightly different. We call this an *inexact match*. Inexact matches occur when two class types are similar but not enough so for the compiler to recognize. This is a good case for refactoring into a bridge, as opposed to inheriting directly.

Below we see our implementor class types. Notice the method `ProcessRequest()`. This method, as we will see in our next code example, is mirrored in the bridge class, but with a slightly different name. For this example, we don't want to change any method names, and because of that the two class types cannot inherit directly. The reason for this is that compilers require that method names match in inherited instances.

```
class HttpRequest
{
    public override void ProcessRequest(string request)
    {
        .......
    }
}
class ISAPIRequest
{
    public override void ProcessRequest(string request)
    {
        .......
    }
}
```

Here we see the class we have targeted to turn into our bridge. Notice this class has a method similar to the implementation class but named differently. For this example, the method's internal functionality is similar, and so meets the requirements of the Bridge pattern:

```
class RequestHandler : Request
{
    public override void Process(string request)
    {
       //some implementation code
    }
}
```

What we wish to do is to make the `RequestHandler` class use our implementor handler types, without the overhead of actual inheritance from these types. Next, we will see how we can accomplish this and not have direct inheritance using the `RequestHandler` class as a bridge.

Solution: Create a series of classes and allow a bridge class to hold the desired instance of the series and adapt its methods, properties, and events through the bridge's methods, properties, and events

The first step after deciding to use the pattern in our refactoring effort is to identify the classes that we wish to make implementors and the classes that will act as the bridge. Since we have done this in our problem section, we will start by refactoring our implementors. This is usually not a difficult task, as they only need to inherit from a common base.

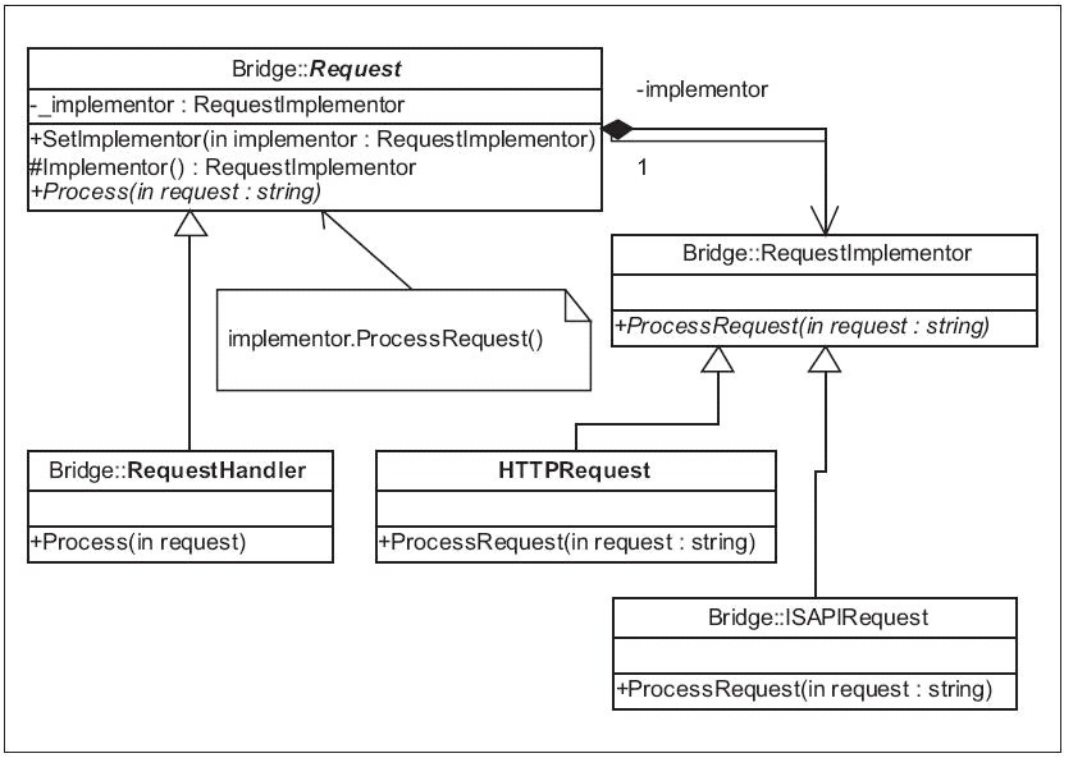

Figure 4-4: UML for Bridge pattern example

Below, we add a base and make both implementors inherit from that base. This allows for the concrete instance, which we will house inside the bridge class. We provide an abstract method for inherited types to share: `ProcessRequest()`. This enables this method to be called from any concrete instance we select inside our bridge.

```
abstract class RequestImplementor
{
    public abstract void ProcessRequest(string request);
}
```

Next, we refactor our implementor classes to have the abstract `RequestImplementor` class as their base. We override the `RequestImplementor` class's abstract method `ProcessRequest()`, which gives us our polymorphic implementation:

```
class HttpRequest : RequestImplementor
{
    public override void ProcessRequest(string request)
    {
        ......
    }
}
class ISAPIRequest : RequestImplementor
{
    public override void ProcessRequest(string request)
    {
        ......
    }
}
```

Next we need to refactor our bridge's base class. We have determined that the `Request` class is a good case for this. We provide a private instance variable for the abstract implementor. We provide a *getter* and a way to set the actual concrete implementation of our abstract implementor to the instance variable. We also provide an abstract method, `Process()`, which will be used to allow the inherited instances of `Request` to have a common method. This method is refactored from the original method.

```
abstract class Request
{
    private RequestImplementor _implementor;

    public void SetImplementor(RequestImplementor implementor)
```

```
    {
        _implementor = implementor;
    }

    protected RequestImplementor Implementor
    {
        get{return _implementor;}
    }

    public abstract void Process(string request);

}
```

In `Request`'s inherited classes we will then override the `Process()` method. Inside the method we will call the instance class of the implementor and pass our arguments from the `Process()` method to the implementor's desired method to bridge to — `ProcessRequest()`:

```
//Refined Abstraction
class RequestHandler : Request
{
    public override void Process(string request)
    {
        Implementor.ProcessRequest(request);
    }
}
```

Thus we have performed a complete refactoring of the implementor into our bridge class without using direct inheritance. Now we can call our bridge, set the desired implementor, and call the `Process()` method, which calls the implementor's intrinsic method `ProcessRequest()`. This gives us a complete bridge between classes without the need for any direct inheritance.

```
Request request = new RequestHandler();
request.SetImplementor(new HttpRequest());
request.Process("This is a HTTP request stream..");

request.SetImplementor(new ISAPIRequest());
request.Process("This is a ISAPI request stream..");
```

Comparison to Similar Patterns

The Adapter pattern is most like the Bridge pattern, in that it uses a host class or facade class to house an instance of the actual desired class, calling the housed methods, properties, and events and hiding these events inside the corresponding methods, properties, and events that exist as public methods for the adapter. The main difference in the two patterns is the adapter's lack of abstraction and inheritance in regard to its internal implementation and the outside adapter. A Bridge pattern might be more useful if several different types of classes or implementations are required for a particular problem. Proxies and facades are also very similar to bridges in that they house some functionality that external sources could not use easily. They basically act as an interface or adapter for subsets of functionality, expanding on the control of the scope and domain from a single class or class type as in the bridge and adapter to more complex subsets of classes and expanded domains.

What We Have Learned

The Bridge pattern is a useful pattern for making slightly incompatible classes and their subclasses more compatible. It gives us a way to do this without using inheritance and by minimizing the abstracted functionality between classes to expected methods, properties, and events. It takes a series of abstracted class types and adapts their functionality to another series of unrelated classes with a minimum of actual cohesion between the two class structures, while providing all the functionality between the two classes that is desired.

Related Patterns

- Adapter pattern
- Facade pattern
- Proxy pattern

- Template pattern

Composite Pattern

What Is a Composite Pattern?

The *Composite* pattern is a collection pattern that allows you to compound different subsets of functionality into a collection and then call each subset in turn in the collection at a given point.

Let's say you had a collection of objects, and each object had a particular method that you wanted to call in series. You could call each method in the collection separately, but this would take time and a lot of code:

```
obj1.Operation();
obj2.Operation();
obj3.Operation();
obj4.Operation();
obj5.Operation();
obj6.Operation();
obj7.Operation();
...... and so on
```

Instead, a composite allows you to call the classes with a single method and loop through the collection, calling the method on each class. This remains flexible because each class in the collection as well as the collection class itself all derive from the same base class. We differentiate between them by their function.

The Composite pattern has three main components. The base class that the other classes share is called the *Component*. This is the class with the common functionality for all classes. The *Leaf* is the object that makes up the individual objects that exist within the collection. The *Composite* is the class that forms the collection object itself. Both the leaf and the composite classes inherit from the component. This is done so that whether you call the individual leaf classes or the composite class, you still call the same method(s).

Suppose you wanted to make calls to a particular method on the class but, depending on whether the class was of type leaf or type composite, you wanted the method to work differently. If you called the method on a leaf, then you only executed the code in that leaf class. But, in the case of the composite class, when you call the method, the composite loops through all the leaf objects inside the composite and calls the method on each one. This allows you to use either a single object's method(s) or get a compounded effect for a whole collection of classes by calling all the leaf classes in the collection.

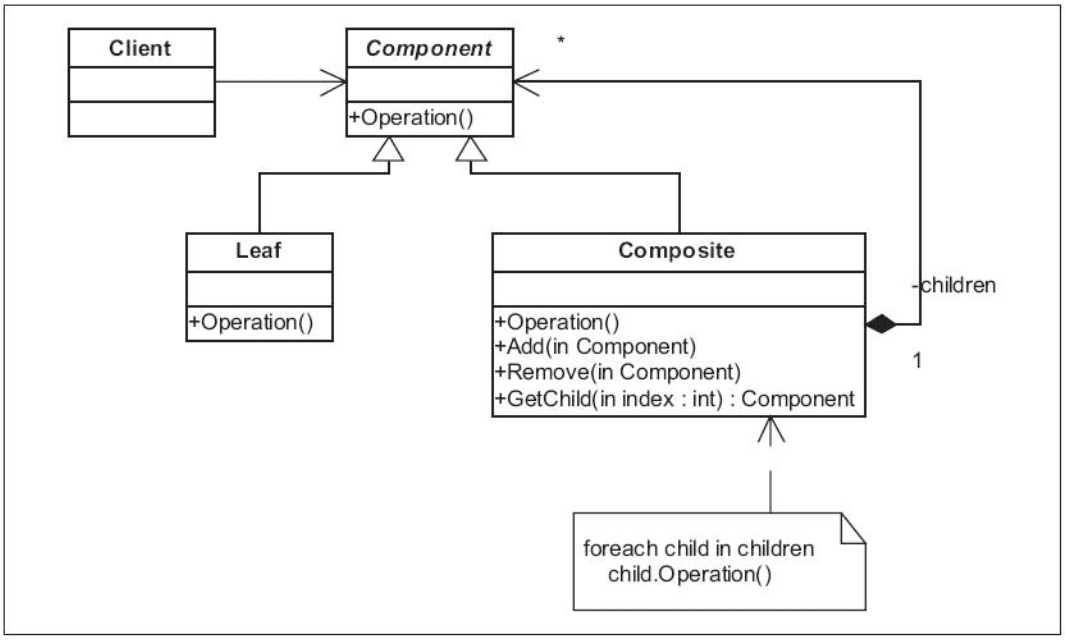

Figure 4-5: UML for Composite pattern

Problem: A way is needed to allow a group of classes with a common method and different functionality to

work together in sequence without compiling the sequence beforehand

Our functional problem is a simple one. We start with a group of objects that each perform a different graphical function. We don't have a streamlined way to connect each one, but we need each object to be connected in a specific fashion and have each object call its `Draw()` method in the proper sequence.

```
//create all objects
OuterPanel root = new OuterPanel();
Circle circle = new Circle();
Square square = new Square();
Triangle triangle = new GraphicalComponent();
InnerPanel child = new InnerPanel();
Line line = new Line();

line.Draw();//first we need to draw the innermost object
child.Draw();//next the InnerPanel
//now we draw our remaining three graphical objects
circle.Draw();//Draw the circle object
square.Draw();//Draw the square object
triangle.Draw();//Draw the triangle object
root.Draw();//finally draw the outermost object
```

As you can see, this is not really very flexible. We have to make a lot of code changes to change the way the objects display. You may also notice there is no clear relationship between each object. So we may have trouble defining which object contains another object, and how the inner and outer objects affect each other. In other words, this is not a great model for object relationships. We see some of the classes appear to be named in such a way as to suggest they might be parents for other objects. This may be handled as hard coded inside each class's `Draw()` method, but is not intuitive based on the apparent class relationships.

In turn, we don't want to have to call the same method on each class over and over. We want the classes containing objects to be able to handle calls to their `Draw()` method, and also use the same method on each of their children.

Solution: Group each class into a composite, which will manage when each class will have its common method called and will allow manipulation of the sequence of calls

We have identified in our problem that we have a need for defining a relationship between parent and child classes. What we need is a way to encapsulate the child classes in parents, and have the parents responsible for how they interact with their children. We can assume each parent may have many children, so we will allow the parent objects to assume the role of a collection. It seems all the objects have a similar function, the `Draw()` method, so we will want a common base class for both the parent and child classes that defines this method.

The first step we will take is to define that base class. This base in the Composite pattern is the component. The component will have an abstract `Draw()` method, as well as any other needed methods, properties, or events to support the base class. We derive the common elements for the class from finding out each class's common elements and putting them into the base. For this case, since the `Component` class will be the base for the leaf collection objects, we will need to implement the basic equals implementation needed for making indexing and comparisons in a collection object. We also want to have a private variable to capture the leaf object's name and a public property for this variable. We also add an abstract method, `Draw()`, which will be implemented in different ways in each of the inheriting subclasses.

```
//Component
abstract class Component
{
    private string _name;

    public Component(string name)
    {
        _name = name;
    }
    public string Name
    {
        get{return _name;}
    }
    public abstract void Draw();

    public override int GetHashCode()
    {
```

```
        return _name.GetHashCode ();
    }
    public override bool Equals(object obj)
    {
        if(obj != null && obj is Component)
            return _name.Equals (((Component)obj).Name);
        else
            return false;
    }
    public override string ToString()
    {
        return _name;
    }
}
```

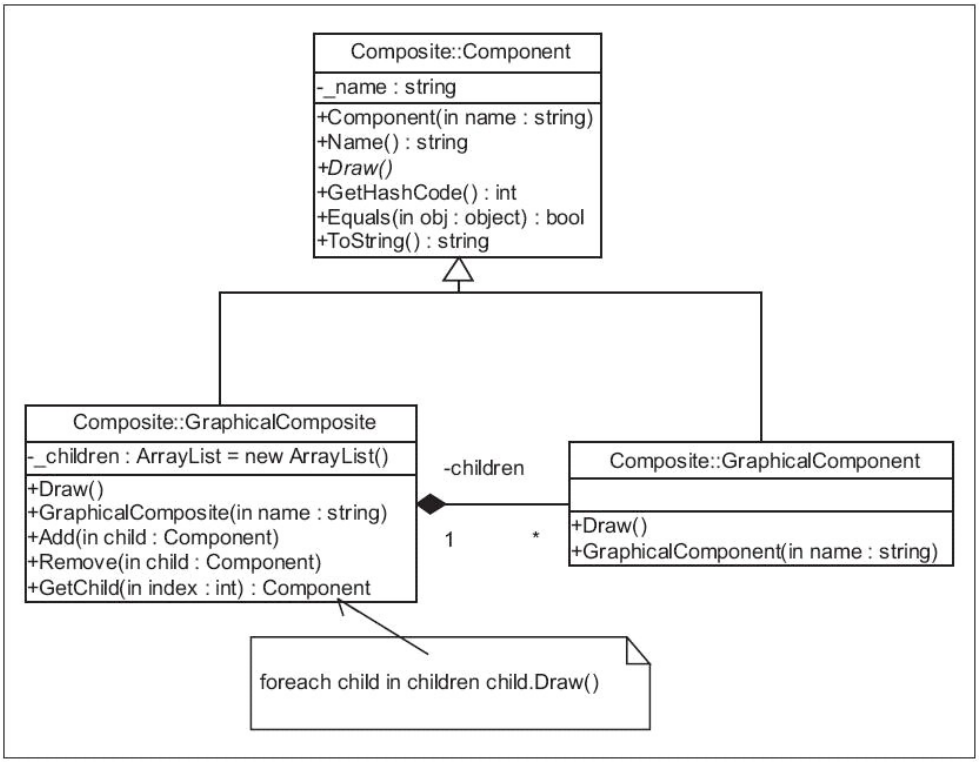

Figure 4-6: UML for Composite pattern example

Next we want to define our leaf class. Since this class will be our class for individual functionality, we will have the overridden `Draw()` method actually do something. In this case it will draw the graphical object:

```
//Leaf
class GraphicalComponent : Component
{
    public GraphicalComponent(string name) : base(name){}
    public override void Draw()
    {
        ......
    }
}
```

Lastly we create our composite object. The first thing we want to do is define a private collection variable to hold our leaf classes:

```
//composite
class GraphicalComposite : Component
{
    private ArrayList _children = new ArrayList();
```

We add a constructor, which takes the name of the component, along with methods to add and remove leaf objects from the collection:

```
public GraphicalComposite(string name) : base(name){}

public void Add(Component child)
{
    _children.Add(child);
}

public void Remove(Component child)
{
    _children.Remove(child);
}
```

We also add a way to feed an index into the collection and get back an individual leaf object. This might be useful if we just wanted to activate one object's `Draw()` method, instead of all of the objects contained in our composite.

```
public Component GetChild(int index)
{
    return (Component)_children[index];
}
```

And the last method we implement is our overridden `Draw()` method. Notice that this method has a looping algorithm to call each child leaf object's `Draw()` method in sequence. So if you call the parent class's method, then you also call all its childrens' methods for `Draw()` as well.

```
public override void Draw()
{
       foreach(Component comp in _children)
           comp.Draw();
    }
}
```

You are not limited to having leaf objects in your composite object. A composite class can also contain other composite classes, which also might hold a number of leaf classes. Although cardinality is not required by the pattern so that each parent and each child knows about itself, you could make this happen. In the basic case, you only have a one-sided relationship in that the composite always knows about its leaf objects, but the leaf objects do not know anything about their parent composite object.

When we now look at the implementation of our new Composite pattern, the first thing we notice is the apparent relationships that we create between our leaf and parent composite objects. When we add leaf classes to the outer composite object, this object actually contains them, establishing the parent-child relationship:

```
GraphicalComposite root = new GraphicalComposite("OuterPanel");
root.Add(new GraphicalComponent("Circle"));
root.Add(new GraphicalComponent("Square"));
root.Add(new GraphicalComponent("Triangle"));
```

We can also add another composite to our outermost one. As you see below, we create a second composite, add a leaf class, and then add the second composite to the first one. This provides a relationship down through the tree to all the leaves contained within the tree, including the ones contained in the other composites.

```
GraphicalComposite childComposite = new
         GraphicalComposite("InnerPanel");
childComposite.Add(new GraphicalComponent("Line"));
root.Add(childComposite);
```

As with any collection class, we can add and remove leaves and composites from any composite we wish. Since we defined the `Add()` and `Remove()` methods by a specific type (the `Component` class), to add or remove an object from the composite we only need to inherit from that type to add to the composite's underlying collection.

```
GraphicalComponent removable = new GraphicalComponent("Single Line");
root.Add(removable);
root.Remove(removable);
```

Now when we call the `Draw()` method on the outermost class, it in turn calls the same method for all the leaf objects contained within. It also calls the method on all the composites it may contain, which in turn will call their children, and so forth, until all classes in the tree have had their `Draw()` method called:

```
root.Draw();
------------
Test for Composite
```

```
Create Root 'OuterPanel' Composite..
Add 'Circle' to Root 'OuterPanel' Composite..
Add 'Square' to Root 'OuterPanel' Composite..
Add 'Triangle' to Root 'OuterPanel' Composite..

Create Child 'InnerPanel' Composite..
Add 'Line' to Child 'InnerPanel' Composite..
Add Child 'InnerPanel' to Root 'OuterPanel' Composite..
Add Child 'Single Line' Component to Root 'OuterPanel' Composite..

Remove Child 'Single Line' Component to Root 'OuterPanel' Composite..

OuterPanel.Draw() called to draw GraphicalComposite.
--Circle.Draw() called to draw GraphicalComponent.
--Square.Draw() called to draw GraphicalComponent.
--Triangle.Draw() called to draw GraphicalComponent.

InnerPanel.Draw() called to draw GraphicalComposite.
--Line.Draw() called to draw GraphicalComponent.
```

Comparison to Similar Patterns

As with any relational pattern, we have the ability to establish relationships between objects on a one-to-one or one-to-many basis. The Composite pattern has relationships to its other objects similar to many other relationship patterns. Just as the Chain of Responsibility can call each of its contained objects in turn, so can the Composite. The Composite pattern also can use one or all of the collection properties of the Iterator pattern, especially if you wish to control the rate of how your composite deals with each class in the collection. The Composite pattern is also similar to the Visitor and Flyweight patterns in that passed-in objects influence the overall Composite's behavior much as they do in both of these patterns. Decorators also have this effect, albeit the structure of a Decorator is much different than the Composite, and it is not a collection. We can use Decorators with the Composite pattern to provide another layer of flexibility to how the controls operate.

What We Have Learned

Using a Composite pattern is an intuitive way to define relationships between classes. It is useful in simplifying method calls when dealing with many objects in a collection. The pattern allows passed-in objects to modify overall behavior, giving a compounded effect for all the objects in the collection, as well as all underlying collections.

Related Patterns

- Chain of Responsibility pattern
- Command pattern
- Decorator pattern
- Flyweight pattern
- Iterator pattern
- Visitor pattern

Decorator Pattern

What Is a Decorator Pattern?

One of the most commonly used structural patterns used in GUI (graphical user interface) design is the *Decorator* pattern. The Decorator pattern gives us a way to use inheritance as a conditional add-on. What does this mean? It means that inheritance is used in the pattern in such a way as to make each inherited class be a sum of all the parts.

The Decorator pattern has four main parts: the *Component*, *Concrete Component*, *Decorator*, and *Concrete Decorator*. The component is usually an abstract class that holds the base functionality for both the non-decorated classes as well as the decorated ones. By non-decorated I mean without applying a decorator class to the existing component. The next aspect of the pattern is the concrete component. This is the non-decorated implementation class of the component, which

we can instance without a decorator. The next part of the pattern gives us the name of the pattern. It consists of two parts: the decorator and the concrete decorator. The decorator is the abstract class that inherits from the component and holds an encapsulated instance of our desired concrete component. The concrete component is the implementation class with the added functionality desired for our decorator.

After reading the description above you may still be wondering how the pattern is useful. One use is to provide flexible inheritance. We can cast our decorator to be of the base type of component, which can give us all the same basic functionality we might have in a non-decorated control, but allows us to add desired functionality in an ad-hoc fashion. By making a control a decorator we add all the control's functions and a few desired new ones, or, if we wish not to have the new functionality, we can simply use the non-decorated control. This interchangeability gives us greater flexibility when dealing with inherited classes.

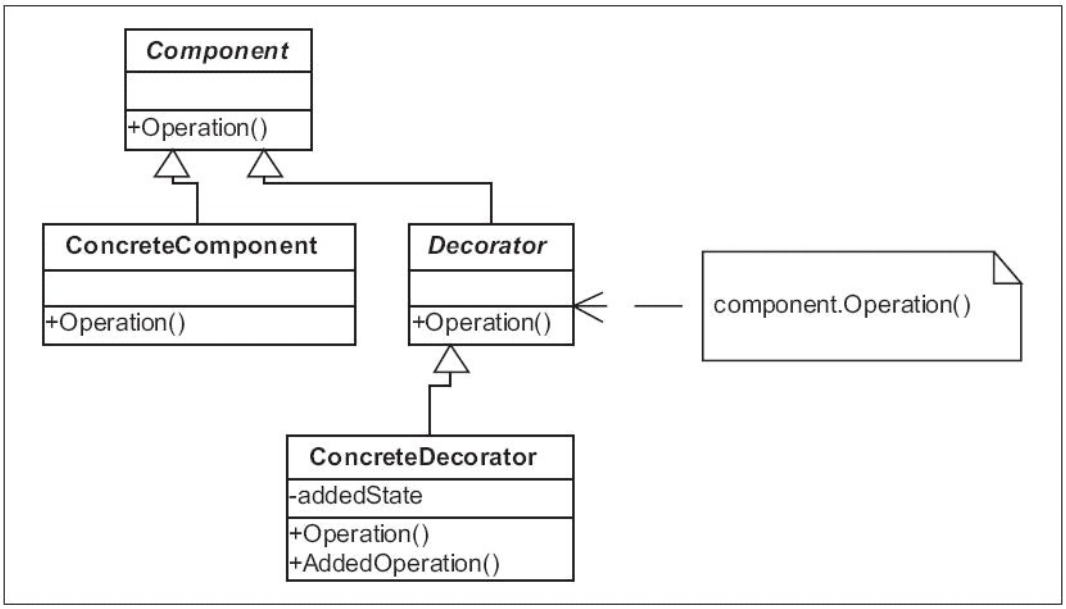

Figure 4-7: UML for Decorator pattern

Problem: We wish to have certain functions associated with a class type only when desired, but not all the time

Our functional problem involves a common user interface control inherited from an abstract base class. We would like at certain times to add functionality to the class, in this case scroll bar logic, but there are times when adding this logic is unwieldy or improper.

We start by looking at our abstract base class called `WindowControl`. We notice that it contains a lot of generic control sizing and positioning properties as we might expect from a base graphical control. The control also contains an abstract method called `Render()`, which we can assume uses the other properties to make the control visible.

```
abstract class WindowControl
{
    private int _height;
    private int _width;
    private int _x;
    private int _y;

    public int Height
    {
        get{return _height;}
        set{_height = value;}
    }
    public int Width
    {
        get{return _width;}
        set{_width = value;}
    }
    public int XCoord
    {
```

```
        get{return _x;}
        set{_x = value;}
    }
    public int YCoord
    {
        get{return _y;}
        set{_y = value;}
    }

    public abstract void Render();
}
```

We already have this class in our code and it is functioning properly. We also want to take a look at the specific concrete implementation class to which we wish to add a decorator. The `TextBox` class inherits our abstract base and overrides the `Render()` method to perform the actual work of rendering the graphical aspects of the control. It takes in its constructor the properties necessary to perform this functionality. It also houses another instance variable, which does not have any effect on the rendering of the control.

```
class TextBox : WindowControl
{
    private string _value;

    public TextBox(int height, int width, int x, int y)
    {
        this.Height = height;
        this.Width = width;
        this.XCoord = x;
        this.YCoord = y;
    }

    public string Value
    {
        get{return _value;}
        set{_value = value;}
    }
    public override void Render()
    {
        Console.WriteLine("--TextBox Height:"+this.Height);
        Console.WriteLine("--TextBox Width:"+this.Width);
        Console.WriteLine("--TextBox X Coord:"+this.XCoord);
        Console.WriteLine("--TextBox Y Coord:"+this.YCoord);
        Console.WriteLine("--TextBox Value:"+_value);
    }
}
```

When we run the code it renders the `TextBox` class as expected. But there are times our text box needs some scroll bar logic added to it:

```
private int _scrollBarWidth;
private int _scrollBarPosition = 0;

public int ScrollBarWidth
{
    get{return _scrollBarWidth;}
    set{_scrollBarWidth = value;}
}
public int ScrollBarPosition
{
    get{return _scrollBarPosition;}
    set{_scrollBarPosition = value;}
}
```

We could just add this logic to the `TextBox` class, but it might be confusing or cause problems when using a non-scrollable text box. Also, because we don't want to render the scroll bar at all times with the text box, adding this functionality to the class would not be advisable. In short, it is not appropriate to add the scroll functions directly to the text box.

Another way we could accomplish this is to simply inherit from the `TextBox` class and create a scrollable `TextBox` class.

This is not hard in simple implementations, but as complexity increases in the bases and across the inheritance chain, this can become problematic.

How do we confront this problem? Let's look at our solution section to see.

Solution: We allow a decorator that inherits from the class we are using to take its place and act as the class, but with the increased functionality of the decorator

When we look at our problem we easily could say we might add Boolean logic (`if...then...else`) to the render method to check the scroll values and render only if values were set:

```
public override void Render()
    {
       if(_scrollBarWidth > 0 && _scrollBarPosition > 0)
       {
       Console.WriteLine("--Scroll Bar Width:" + _scrollBarWidth);
       Console.WriteLine("--Scroll Bar Position:"
                         + _scrollBarPosition);
       }

       Console.WriteLine("--TextBox Height:"+this.Height);
       Console.WriteLine("--TextBox Width:"+this.Width);
       Console.WriteLine("--TextBox X Coord:"+this.XCoord);
       Console.WriteLine("--TextBox Y Coord:"+this.YCoord);
       Console.WriteLine("--TextBox Value:"+_value);
    }
```

But that would not really make use of OOP principles in our code or be intuitive when making changes. We can't quickly tell by this code whether or not we have a scrollable text box. We have to wait until run time and then try to see if valid values are present. This can be both confusing in the intent of the object as well as in violation of the rules of encapsulation: our class should only know about scroll bars if it really does have one.

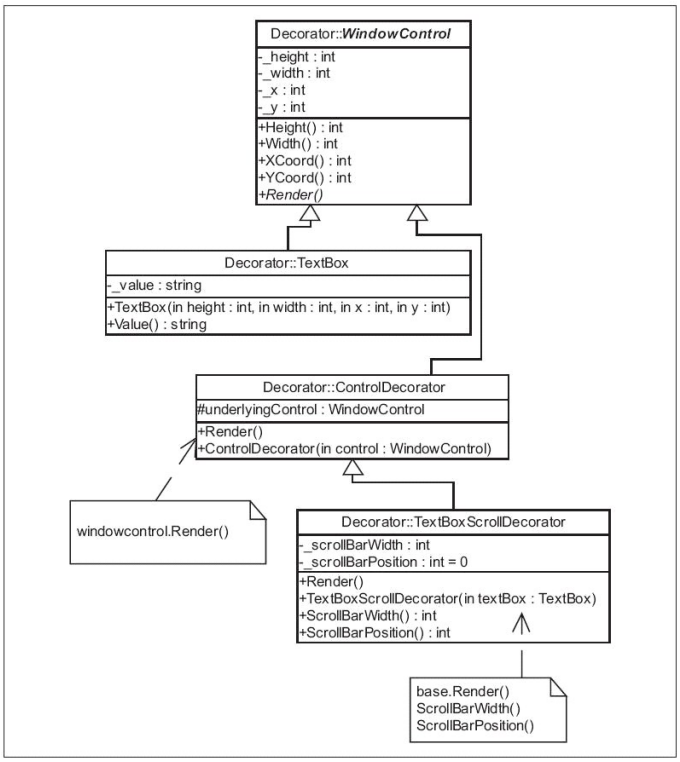

Figure 4-8: UML for Decorator pattern example

So we need another way to deal with this situation. A Decorator pattern seems to be the way to go. Using a decorator we can interchange our objects. We can use the decorated class when we need a scroll bar, and our `TextBox` class when we don't. Let's begin the refactoring effort at the point of creation of our decorator.

As we saw in our problem section, we already have the component and concrete component. For this refactoring effort, we don't wish to change either class. The true strength of the Decorator pattern comes to light in this. Since we don't have to modify existing functionality, we can easily start on the new aspects of code we wish to procure.

We start with our abstract decorator class. Following the pattern, we inherit this class from the abstract component we already have: `WindowControl`. We do this to provide the base functionality we might get with the `TextBox` class. Next, we need to provide a protected instance of our component. We do this so that a concrete instance can be passed and we can take advantage of that instance in our decorator. What we want is to capture all the functionality of our concrete component, while encapsulating it as an instance variable. Notice we also override the abstract method `Render()` and call our protected instance's `Ren der()` method inside. The instance is made protected so that the inherited concrete decorator can manipulate it directly.

```
abstract class ControlDecorator : WindowControl
{
    protected WindowControl underlyingControl;

    public ControlDecorator(WindowControl control)
    {
        underlyingControl = control;
    }
    public override void Render()
    {
        underlyingControl.Render();
    }
}
```

Next, we create our concrete decorator. This class is where we can directly add and change the functionality we get with our `TextBox` class. Generally, we want to add our new functionality of scroll bar logic to the existing `TextBox` functionality. We inherit from the abstract base decorator class, and provide our instance variables for the scroll bar settings:

```
class TextBoxScrollDecorator : ControlDecorator
{
    private int _scrollBarWidth;
    private int _scrollBarPosition = 0;
```

The constructor for the class takes as its input parameter the `TextBox` object type. This is done to provide a more intuitive way to determine that we are decorating a `TextBox` class:

```
public TextBoxScrollDecorator(TextBox textBox) : base(textBox)
{
}
```

Next, we add our properties to set the scroll bar:

```
public int ScrollBarWidth
{
    get{return _scrollBarWidth;}
    set{_scrollBarWidth = value;}
}
public int ScrollBarPosition
{
    get{return _scrollBarPosition;}
    set{_scrollBarPosition = value;}
}
```

The last step is to override the `Render()` method and add our scroll bar rendering logic to it. Notice we call our base method first. We do this because in the rendering of the control to the user interface, the base text box component needs to be rendered before the scroll bar. This can be changed on a case-by-case basis.

```
    public override void Render()
    {
        base.Render();
        Console.WriteLine("Added decorator values:");
        Console.WriteLine("--Scroll Bar Width:" +_scrollBarWidth);
        Console.WriteLine("--Scroll Bar Position:"
                          +_scrollBarPosition);
    }
}
```

Now we have all the necessary pieces to use our decorator. We need to look at how we can use the decorator and text box as interchangeable pieces. In our code we have need of a basic text box. We can instantiate this, set its properties, and call its `Render()` method:

```
TextBox textBox = new TextBox(250,350,1200,300);
textBox.Value = "Some Text...";
textBox.Render();
```

Next, in some other part of our code we need to make our text box scrollable. We need to use a type that inherits from the component, which our decorator does. We feed our previous text box instance to the decorator, set the scrollable components, and call its `Render()` method. The method on our decorator calls the encapsulated text box instance's method, and then adds its decorated render logic to it, giving a composite effect.

```
TextBoxScrollDecorator scrollable = new
       TextBoxScrollDecorator(textBox);
scrollable.ScrollBarPosition = 20;
scrollable.ScrollBarWidth = 10;
scrollable.Render();
```

The decorator implementation now gives us an intuitive and easily recognizable and interchangeable way to deal with adding functionality, without using direct chains of inheritance. We minimize confusion by decreasing the levels of inheritance between the `TextBox` class and any inherited classes we might otherwise create.

Comparison to Similar Patterns

The Decorator pattern is readily comparable with other patterns. One such pattern is the Adapter pattern. This pattern also hides another class's functionality within, and allows only an expected access to the inner class. Decorators often work well to increase the flexibility of Composites, where class relationships extend through a collection and methods can be called between several classes to create a composite whole.

What We Have Learned

Decorators give us a more flexible way to deal with class inheritance. They allow us to extend and interchange class types and functionality by allowing us to add new functionality to an existing class in a more easily recognizable and maintainable way. It gives us more explicit control of a class's functionality, without adding to the complexity of the code when using levels of inherited classes.

Related Patterns

- Adapter pattern
- Composite pattern
- Visitor pattern

Facade Pattern

What Is a Facade Pattern?

A *Facade* pattern allows grouping of subsystems behind a unified interface to allow a central access point to these subsystems. If you desired to limit the access to a group of subsystems or define a limited interface to these subsystems, you might use a facade. A facade is a way to control the access to these subsystems.

The Facade pattern has one main component: the *Facade Interface*. The facade interface may consist of several parts, but it is considered one component. This interface is the hub for accessing controlled subsystems inside an assembly or package.

There are many ways and reasons to use a facade. API layers are good examples of where and how to use a facade. An API layer allows certain functionality to be visible to code bases outside the API, but only allows specific interfaces to be used to access any code inside the API layer.

For instance, if you had a group of classes that together defined a particular code flow that made up a process, but calling these classes out of order would result in catastrophic failure of these subsystems, you could make all these classes internal to their assembly. This would hide these classes from access outside the assembly. You then could create a

facade to encapsulate the calling order of each class, thereby controlling code flow behind the facade. By making each class internal to the assembly, you guarantee that you do not allow unexpected access to the classes. Inside the assembly you can decide when and where each class is called. Methods on the facade interface then could be provided to access the subsystem classes in a controlled fashion. No outside access to these classes would be permitted and only the code internal to the facade could decide the class's calling order, thus preventing a catastrophic failure. The facade could then allow controlled access to each class in its proper calling order, encapsulating the code flow of the entire subsystem structure.

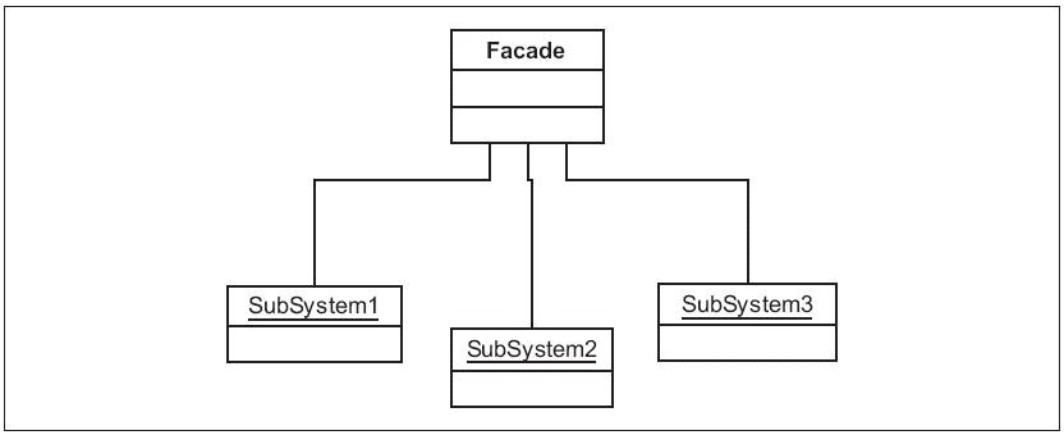

Figure 4-9: UML for Facade pattern

Problem: We have several classes whose methods make up a process, but we need to control the order in which these classes are called by code outside the assembly

For our functional example, we have a group of classes that reside inside an assembly that we want to make into an API (application program interface). Each class has functionality specific to a process flow, and together when called in the right order they produce the desired code implementation. But you have discovered that these classes are being called in code bases outside the one your API classes are in that cause a catastrophic failure in your API layer. You need some way to control how, when, and in which order each class is called. You also want to provide a simpler interface for code outside the assembly to call these classes. In short, you need a way to provide a single interface to access all the needed classes and their methods to accomplish a task.

```
...called from outside the API assembly

BusinessRules rules = new BusinessRules();
rules.GetRules();

BusinessDAL dal = new BusinessDAL();
dal.Update();

//calling the validation after the DAL update fails
BusinessValidation validation = new BusinessValidation();
validation.CheckIsValidRequest(updateValue);
```

Solution: Create a facade layer with interfaces to desired classes that are called in the proper order and allow the subsystem classes to only be called inside the facade assembly

Your solution is to provide a facade interface. The facade will allow you to control and limit the access to each subsystem class. It will also allow you to encapsulate the calling order or *order of operation* of the class group.

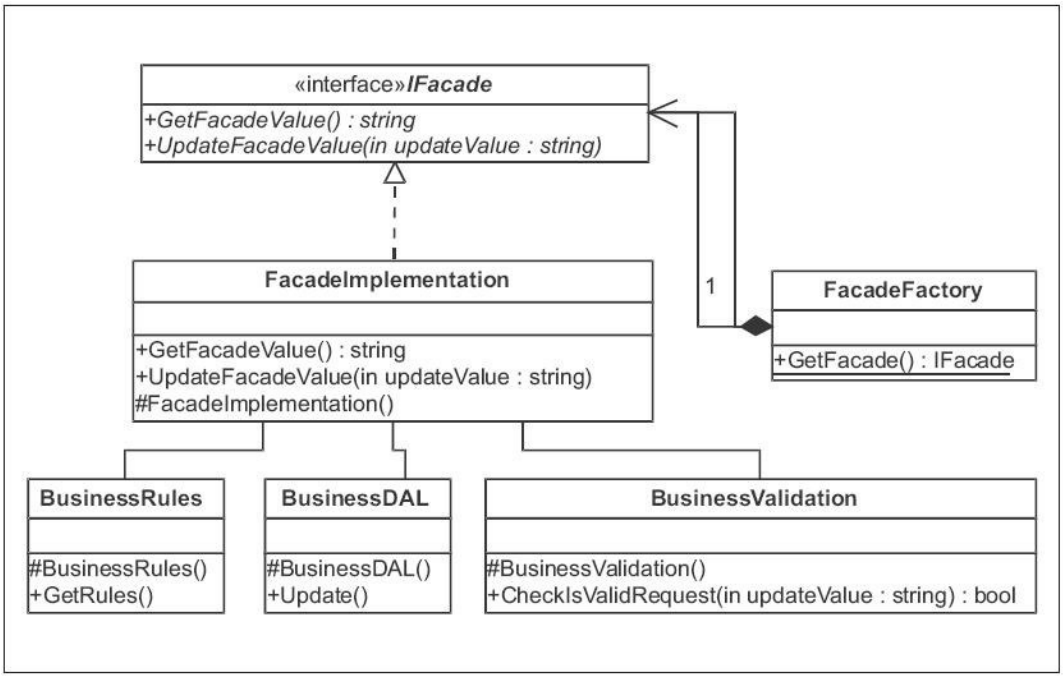

Figure 4-10: UML for Facade pattern example

The first step in implementing the Facade pattern is to make your subsystem classes inaccessible to any outside assembly. To do this in .NET you need to change the access modifier to `internal` for each class declaration. You also need to change the constructor to `protected internal` to guarantee that no access outside the current assembly is possible.

```
internal class BusinessDAL
{
    protected internal BusinessDAL(){}

    public void Update()
    {
        ...
    }
}

internal class BusinessRules
{
    protected internal BusinessRules(){}
    public void GetRules()
    {
        ...
    }
}

internal class BusinessValidation
{
    protected internal BusinessValidation(){}
    public bool CheckIsValidRequest(string updateValue)
    {
        ...
    }
}
```

Notice that each method above is public. Because the class is marked `internal`, the access to these methods is limited to instantiation of the class only inside the assembly or package. For Java implementations, you need not add any class access modifier; just declare the class thus:

```
class BusinessDAL {...}
```

This sets the access to internal to the package.

Now that you have all your subsystem classes in your assembly set to inaccessible, you need to build the external access interface you will use for the facade implementation:

```
public interface IFacade
{
    string GetFacadeValue();
    void UpdateFacadeValue(string updateValue);
}
```

For this example, we are allowing two methods to be accessed in the interface. These two methods will point back to our facade implementation class methods. The interface allows disconnected access outside the API layer.

Now we can create the implementation class that will house the facade functionality for each method. We create a class that implements the `IFacade` interface so we can have a place to implement our facade methods:

```
public class FacadeImplementation : IFacade
{
```

We set the constructor of our facade implementation class to `internal` in .NET. We do this in Java by using the keyword `protected`. This way we do not allow the facade to be created outside the assembly. Why would we do this? We want to allow access from outside our API layer to only the interface, not the implementation class of the facade. To do this, we need to limit the creation of the facade implementation class to a factory, which only returns the interface. In this way, we can guarantee that access outside our API layer is only done through the `IFacade` interface.

```
protected internal FacadeImplementation()
{

}
```

Our implementation methods are all set to have public access modifiers. This is done to allow access to the facade methods outside the assembly or package.

```
public string GetFacadeValue()
{
       ...
}


public void UpdateFacadeValue(string updateValue)
{
    ...
}
```

Now let's look at how we control the program flow of the subsystem classes inside the facade methods. For our example, we will look at the method that performs the update, which has several class methods that must be run in order to accomplish a task:

```
public void UpdateFacadeValue(string updateValue)
{
    BusinessValidation validation = new BusinessValidation();
    validation.CheckIsValidRequest(updateValue);
    BusinessRules rules = new BusinessRules();
    rules.GetRules();
    BusinessDAL dal = new BusinessDAL();
    dal.Update();
}
```

Notice each class is created and the order of operations is controlled in an expected fashion. Doing this accomplishes our goal of limiting access and allowing a controlled path through our subsystem code. Now when we call the facade interface from the factory we have a single route into the subsystem, and this route is determined inside the facade.

```
//Facade returned from factory.
IFacade facade = FacadeFactory.GetFacade();

// Facade subsystem update called:
facade.UpdateFacadeValue("some value");
--Validation subsystem called.
--Rules subsystem called.
--Data Access Layer subsystem called.
--Facade subsystems performed an update.
```

This makes it simpler for both the calling application, since it is given a simple access point, and the facade code, whose logical functional path is set and accessed in only one way.

Comparison to Similar Patterns

The Facade pattern is similar to the Proxy pattern in that they both provide access to subsystems. However, the Proxy pattern is better suited to remote access or multi-domain specific access, and the Facade is better suited to API or code layers within the same domain or code base. Adapters and Bridges could also be compared, but their functionality requires less abstraction of their subsystems than a facade suggests, requiring only some way for similar code bases to interact without sharing their underlying classes. Adapters and Bridges also require less structured and comprehensive restriction of the subsystems, usually only spanning the scope of specific classes, while a facade usually controls multiple classes and is always used to limit access to a package or assembly, controlling and defining the access and order of operations of these subsystem classes.

What We Have Learned

Facades are an interesting way to provide controlled access to classes within a package or assembly to code outside the assembly or package. It allows encapsulation of subsystem classes and provides a way to control the order of operations and interaction of these classes inside the assembly while providing a seamless access point outside the assembly.

Related Patterns

- Adapter pattern
- Bridge pattern
- Proxy pattern

Flyweight Pattern

What Is a Flyweight Pattern?

The *Flyweight* pattern allows you to support a large number of granular objects by allowing them to be shared with intrinsic values and stored within a factory. These shared objects can be modified by providing extrinsic values from their individual contexts. The fine-grained objects themselves do not need to know or have reference to the context at all as a rule.

The Flyweight pattern has three main components: the *Flyweight Factory*, the *Abstract Flyweight*, and the *Concrete Flyweight*. There is also the *Unshared Concrete Flyweight*. The flyweight factory acts as a repository for the shared flyweight classes and creates new ones if not already constructed. The abstract flyweight class defines the shared attributes common to any flyweight object of this type. It acts as an interface, which provides methods to receive the extrinsic state from a context. The concrete flyweight class defines and stores intrinsic state. It is shareable and its state must be independent of any context. The unshared concrete flyweight class defines flyweights that are not shared. It can have children that are shared, but since the flyweight interface only allows sharing and does not enforce it, this class is useful when a single unshared instance is required.

So what is a flyweight and what does it do? Well, imagine you had a program that required a large group of object instances to function. To create a different object for each instance would be very costly in regard to both CPU memory and program load. Therefore, instead of creating a different object class for each instance, we allow shared objects that are independent of outside contexts. That is, we can share the objects repeatedly without allowing the intrinsic state to be modified.

Intrinsic state is the state that resides within the shared object and is independent of context. *Context* is the particular placement of each object in code flow, with the particular values and states associated with that context. The *extrinsic state*, or state that is associated with the context, is allowed to pass into the flyweight class without modifying the intrinsic state of the flyweight.

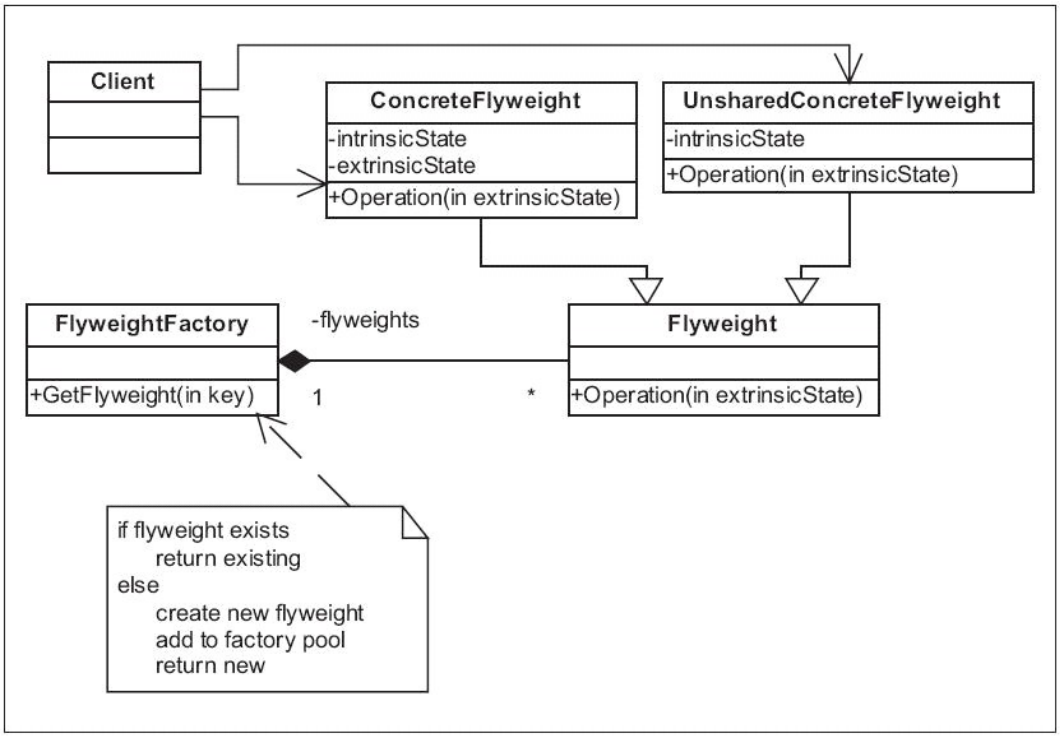

Figure 4-11: UML for Flyweight pattern

Problem: A text document uses individual object instances for each character and quickly becomes unwieldy

For our functional problem, we have a document program that has multiple character objects. These objects have gradually become unwieldy as we have added new characters to the document, and so we need a more efficient model to replace them. Right now, each character is represented by an object. When the user of the program adds another character to the document, it creates a new object and changes the font, size, and other attributes of the object and then adds this object to the document. As the user places more characters inside the document it starts to grow in size, making the program use an exponentially large amount of memory. Each character adds to the memory used by the document program. As the user types in more characters, the program starts to slow:

```
Document.AddCharacter(new Character('C',"Arial",8,"Black",140,100));
....... User types in 300 more characters
```

We need a way to allow each object to only be created once and, depending on its position in the document, have its state modified according to the formatting for that text section.

Solution: Use shared flyweight objects to instantiate a limited number of character objects that use their current context to change the extrinsic state

Our solution to the problem above is to use shared objects instead of creating new ones repeatedly. To accomplish this, we will create small granular objects called flyweights to hold only the basic definitions of a character. Since the characters we will use are used more than once inside our document object, we allow certain formatting to occur depending upon where in the document the character is placed. Each flyweight will have an intrinsic state that exists in the shared object, and methods to allow extrinsic state modification for each context that uses the shared object. The methods that allow extrinsic state to be modified will accept extrinsic state changes through the method's input parameters, but will not store them inside the object. So each call to these methods will only use the passed-in state values inside the method and will lose these values when the scope of the context has passed.

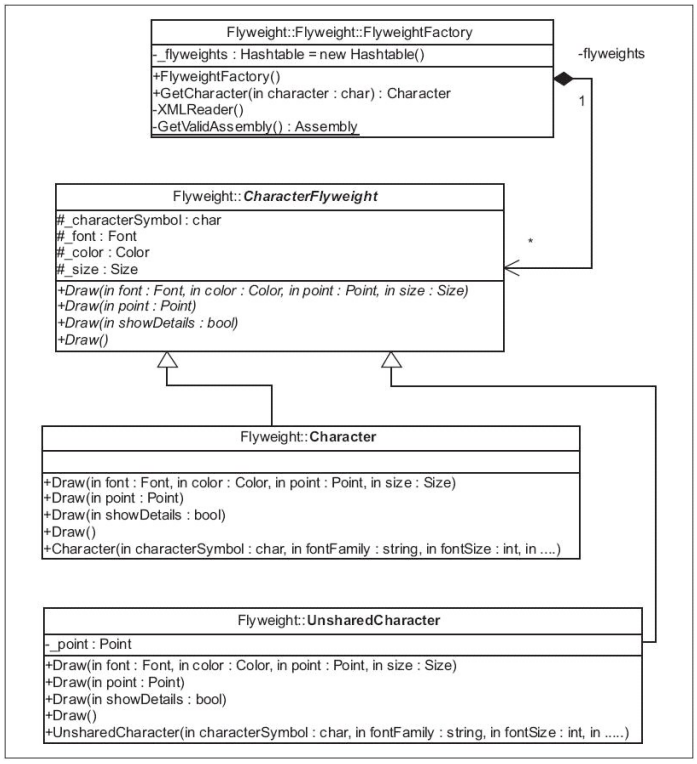

Figure 4-12: UML for Flyweight pattern example

The first step in implementing the pattern is capturing the variables that house the intrinsic state in our abstract flyweight object. We do this by providing them as protected variables accessible to their inherited classes:

```
abstract class CharacterFlyweight
{
    protected char _characterSymbol;
    protected System.Drawing.Font _font;
    protected System.Drawing.Color _color;
    protected System.Drawing.Size _size;
```

Next, we need to define the methods to allow the context to modify the extrinsic state. We use three methods overriding the base `Draw()` method that accept different state complexities. The number and type of methods are dependent on what you might actually need to modify in the extrinsic state inside the flyweight for the current context. Each method you use should allow different extrinsic state complexities to be modified from the default intrinsic state. The actual implementation of these methods should not, however, modify any intrinsic state variables.

```
    public abstract void Draw(
        System.Drawing.Font font,
        System.Drawing.Color color,
        System.Drawing.Point point,
        System.Drawing.Size size);

    public abstract void Draw(System.Drawing.Point point);
    public abstract void Draw();
}
```

Next, we will create our concrete flyweight object. This is the actual shared implementation object. We define the input parameters for the constructor of the class to initialize the intrinsic state of the object. The object will be created in our factory object and the intrinsic shared state variables will be input by the factory upon class creation.

```
class Character : CharacterFlyweight
{
    public Character(
        char characterSymbol,
        string fontFamily,
        int fontSize,
        string color,
        int width,
        int height)
```

```
    {
        _characterSymbol = characterSymbol;
        _font = new System.Drawing.Font(fontFamily, fontSize);
        _color = System.Drawing.Color.FromName(color);
        _size = new System.Drawing.Size(width, height);
    }
```

We implement the abstract flyweight's extrinsic modification methods, allowing the passed-in state to override the intrinsic values only in the current context. These methods will be accessed by the context at run time and will change the way the character is rendered inside the document. The intrinsic values are not replaced or modified in this method because it would change the flyweight's intrinsic values throughout all contexts that are accessing this character throughout the document. Instead, depending on the complexity of state modification desired, the intrinsic values are replaced with the extrinsic ones during the execution of the `Draw()` method. Each method allows different complexities for state modification.

In this instance the method will change all state for the context of the method:

```
public override void Draw(
    System.Drawing.Font font,
    System.Drawing.Color color,
    System.Drawing.Point point,
    System.Drawing.Size size)
{
    .... All intrinsic state is replaced by the extrinsic
         state passed in
}
```

Here the method will replace only certain intrinsic values inside the method with extrinsic ones:

```
public override void Draw(System.Drawing.Point point)
{
    .... Only one intrinsic state value is replaced
}
```

And here the method allows the default intrinsic state to be used, passing in no extrinsic state changes to the method:

```
    public override void Draw()
    {
        .... Default intrinsic state is used unmodified
    }
}
```

Next, we create our unshared flyweight. This class is very similar to the shared class except we require values that are associated directly with the context inside our constructor. We house these context-related values in a variable inside this class that stores the coordinates of the character within the document as a `System.Drawing.Point` object:

```
class UnsharedCharacter : CharacterFlyweight
{
    private System.Drawing.Point _point;

    public UnsharedCharacter(
        char characterSymbol,
        string fontFamily,
        int fontSize,
        string color,
        int width,
        int height,
        int xCoord,
        int yCoord)
    {
        _characterSymbol = characterSymbol;
        _font = new System.Drawing.Font(fontFamily, fontSize);
        _color = System.Drawing.Color.FromName(color);
        _size = new System.Drawing.Size(width, height);
        _point = new System.Drawing.Point(xCoord, yCoord);
    }
```

Inside the `Draw()` method we use the intrinsic `System.Drawing.Point` object to determine the placement within the document object. Since there is one unshared flyweight instance per context location, having intrinsic methods that associate with the context do not interfere with any other implementation of this object within the program.

```
public override void Draw(bool showDetails)
{
    Console.Write(String.Format("Intrinsic State: Font:{0},
                  Color:{1}, X/Y:{2}, Font Size:{3}, Symbol:{4}",
    new object[]{_font, _color, _point, _size, _characterSymbol}));
}
```

The last step in the pattern implementation is the flyweight factory. The factory will act as a creational construct for the individual shared flyweight classes. It will instantiate the class if not already created and store it for any context that wishes to access a flyweight. The factory has one main method called `GetCharacter`. This method returns a fly- weight class creating the class if not already constructed. A hash table inside the factory acts as a storage repository, giving references to the shared flyweight objects to each context requesting an instance.

```
class FlyweightFactory
{
    private Hashtable _flyweights = new Hashtable();

    public Character GetCharacter(char character)
    {
       Character c = (Character)_flyweights[character];
       if(c == null)
       {
           XMLReader();
           c = (Character)_flyweights[character];
       }
       return c;
    }
    private void XMLReader()
    {
        ..... Creates characters identified in the xml config file
            and sets their intrinsic values...
    }

}
```

So now let's look at how the code works. First, we create an instance of our flyweight factory. Next, we call the `GetCharacter` method to retrieve a shared character flyweight object.

```
FlyweightFactory factory = new FlyweightFactory();
Character g = factory.GetCharacter('G');
```

Next, we call one of the extrinsic methods, passing in state variables telling the character object where in the document object to draw the point, what font to use, and so forth. This is what we mean when we say the context. The point in the code where we are when we make our extrinsic call to pass in the current context's values is the location of our current context.

```
g.Draw(new System.Drawing.Font("Georgia", 9),
    System.Drawing.Color.FromName("FloralWhite"),
    new System.Drawing.Point(200, 250),new
    System.Drawing.Size(45, 56));
```

If we look at the test output, we can see the extrinsic values are rendered in the document:

```
Extrinsic State: Font:[Font: Name=Georgia, Size=9, Units=3,
                      GdiCharSet=1, GdiVerticalFont=False],
                      Color:Color [FloralWhite], X/Y:{X=200,Y=250},
                      Font Size:{Width=45, Height=56}, Symbol:G
```

If we wanted to use the shared intrinsic state of the character, we could call the `Draw()` method with no parameters:

```
g.Draw();
```

In the last example above, since we did not modify the intrinsic context by calling the extrinsic method, the values should be the same for every call to the character object. If we looked at the intrinsic state in the test output, we would see that it had indeed not been modified by the previous extrinsic state method call of `Draw()`.

```
Intrinsic State: Font:[Font: Name=Microsoft Sans Serif, Size=9,
                       Units=3, GdiCharSet=1, GdiVerticalFont=False],
                       Color:Color [Black], Font Size:
                       {Width=140, Height=100}, Symbol:G
```

Next, we can call any other character as many times as we wish, setting the extrinsic state or allowing the default intrinsic state to be rendered to the document. We do not suffer a memory penalty in this way because we are reusing each character object repeatedly instead of creating a new character each time one is added to the document. We instead allow the document context to modify the character's position, font, color, and other variables, depending on where in the context we call each shared instance.

```
Character o = factory.GetCharacter('O');
o.Draw();
Character a = factory.GetCharacter('A');
a.Draw();
Character t = factory.GetCharacter('T');
t.Draw();
```

Lastly, let's look at the unshared concrete flyweight and talk briefly about its usage. As we stated above in our description of this object within the pattern, we do not have to have each flyweight object shared. The flyweight abstraction allows sharing, but no enforcement of sharing occurs. This is to allow unshared objects that may contain many shared flyweights as children. This gives us both the option of sharing objects or allowing many instances of objects to be created. This might be useful if we had a character that was only used rarely or needed special context-related conditions to be useful in the document, such as a paragraph element. For our example, we create an unshared flyweight object and pass in our context-related coordinates, font, size, color, and other relevant values:

```
UnsharedCharacter unshared = new
UnsharedCharacter('$', "Arial", 9, "Black", 140, 100, 223, 355);
```

Next, we call the intrinsic method `Draw()`. This method implementation will use our context-related data as passed into the constructor:

```
unshared.Draw();
------------
Test for Flyweight
Spell 'GOAT':
Extrinsic State: Font:[Font: Name=Georgia, Size=9, Units=3,
    GdiCharSet=1, GdiVerticalFont=False], Color:Color [FloralWhite],
    X/Y:{X=200,Y=250}, Font Size:{Width=45, Height=56}, Symbol:G

GOAT

Spell 'GAS':
Intrinsic State: Font:[Font: Name=Microsoft Sans Serif, Size=9,
   Units=3, GdiChar Set=1, GdiVerticalFont=False], Color:Color
   [Black], Font Size:{Width=140, Height=100}, Symbol:G

GAS

Get Unshared Character:
$
```

The particular context of the unshared character object gives us back the intrinsic values from that context.

Comparison to Similar Patterns

The Flyweight pattern uses both the State and Factory patterns to allow its implementation to work. This is one of the better examples of using a factory to manage shared objects, maintaining their state across context calls. This pattern would work well with the Composite pattern to identify parent to child hierarchical relationships inside the document. This kind of tree relationship would also help to group objects for the context. For example, you might use the Composite pattern to house groups of characters to share similar fonts or other attributes. This pattern also resembles the Strategy pattern in that the flyweights are small strategy classes handled by the flyweight factory. Each flyweight, depending on context, could be identified as a separate algorithm and each is interchangeable.

What We Have Learned

Flyweights allow us to control program size and keep it from becoming unmanageable. They allow us to share object instances instead of creating new objects for programs containing very granular and numerous objects. Using a shared or unshared interface, we can allow the current code flow to change the extrinsic attributes related to the flyweights without modifying the intrinsic shared attributes, thereby allowing context changes on objects to occur instead of facilitating the need for unnecessary object creation. This pattern usually works well if numerous object instances would slow down the

system and allowing shared relationships for the context of the program to operate against would be possible.

Related Patterns

- Composite pattern
- Factory pattern
- Interpreter pattern
- State pattern
- Strategy pattern
- Template pattern

Proxy Pattern

What Is a Proxy Pattern?

Proxies generally provide a stub or placeholder for an actual implementation object. It allows the creation, security, or accessibility to be handled outside the current code domain, or latently handled on another address space. There are generally thought to be four main types of proxies:

- A *Remote Proxy* provides a localized stub, placeholder, or interface for an object in a different address space. If you had a program on a different server or on a different domain to which you wished to provide a controlled reference, this type of proxy might come into use.
- A *Virtual Proxy* only creates objects when asked. In other words, if you have an interface to this kind of proxy, it may only create the actual object the interface references when you actually make a method call or direct reference to the item.
- A *Protection Proxy* controls security to the actual object the proxy references. You might use this when an object has different security accessibility than the domain or address of where the proxy might be referenced. Depending on the access point or the level of security from the accessing program, this proxy might allow or fail a call to its referenced object.
- A *Smart Reference* or *Smart Pointer* handles object referencing of the actual object the proxy represents. It can monitor the number of threads accessing the actual object and free resources when necessary. It can load the persistent object into memory when referenced and handle initialization. It can handle transactional locking on the actual object so that other calls to that object cannot change the object while locked.

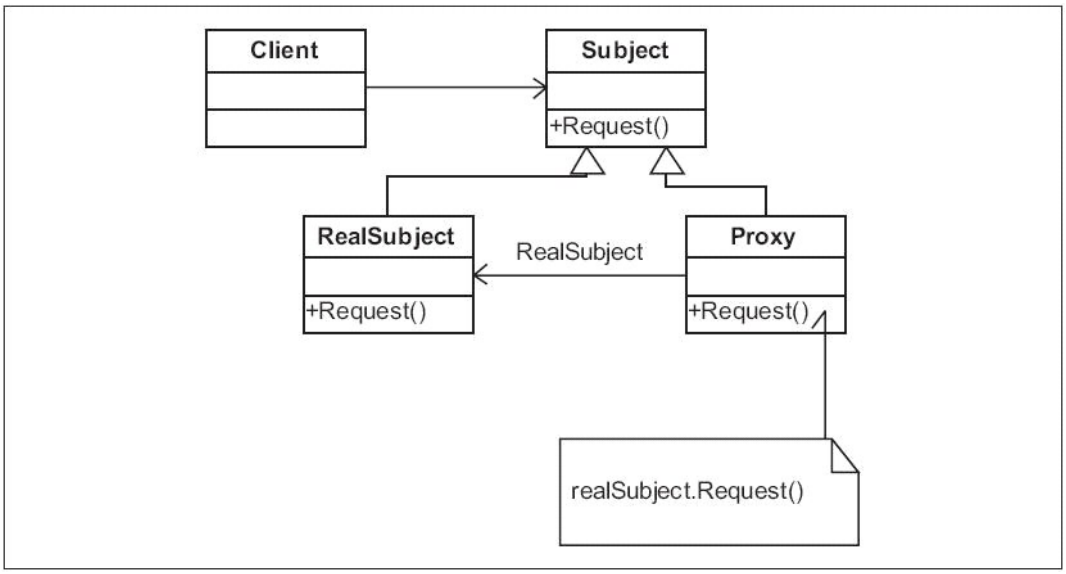

Figure 4-13: UML for Proxy pattern

The Proxy pattern has three main components: the *Subject*, the *RealSubject*, and the *Proxy*. The subject defines the common interface for the real subject and proxy. The real subject is the actual object that the proxy represents. The proxy maintains the reference to the real subject and acts as the access point for the real subject.

The most common usage for a proxy is when you have two incompatible systems or two separate domains and wish to allow these systems or domains to have access points to each other. Another common use for a proxy is to control access to an object to programs outside its domain or address space. Inside the proxy you can manage the access to external domains to any object you desire. The proxy can initialize the actual object and check the security level of the calling program. It can manage multiple instances of the objects or handle the unit of work for the object from outside access points.

Problem: You have an object in a different domain or server address you would like to access in the current domain, but do not want to allow construction of the object in the local domain

For our example, we need to create a way to access a program that can run in its own domain space. It would be too costly to run the program in the current domain space, or the program we wish to run simply resides in another domain space for security reasons. We need to access the program but do not want to load it right away. Currently we create the objects in our current domain. However, we might want several other application domains to access the object, and do not wish each domain to create instances of the object in an unmanaged fashion. We also wish to have a central point for managing instances and security for the actual object's domain. Below, we see the code in our current domain:

```
Subject subject = new Subject();
return subject.GetRequest();
```

For the reasons above we wish to provide a central access point for the `Subject` object's `GetRequest()` method. We wish to hide and encapsulate the security protocols, initialization, and instance creation of the `Subject` object. In addition, we need a way to share desired functionality either remotely or between otherwise incompatible domains.

Solution: Use a proxy to bridge the two separate domains and allow access between programs

We will allow a multitude of different domains to access our proxy object. The proxy object will handle how and when we create our `Subject` object implementation. It will also mask any initialization and the pointer into the domain where the `Subject` object exists.

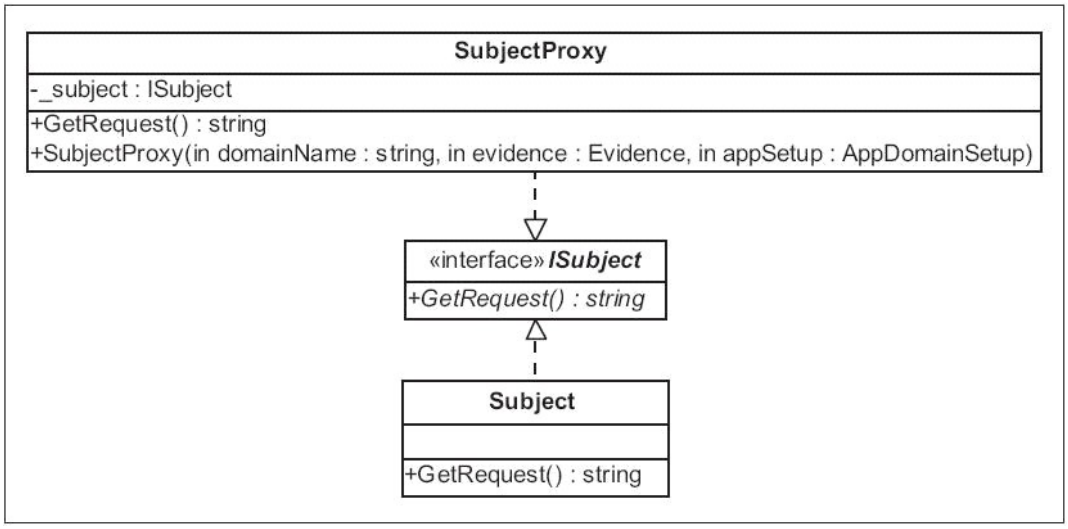

Figure 4-14: UML for Proxy pattern example

The first step in building a proxy around our `Subject` object is to refactor our existing `Subject` object and create the `ISubject` interface. This interface works to define the functionality shared between the `Subject` and `Proxy`:

```
// Subject
public interface ISubject
{
    string GetRequest();
}
```

Next, we refactor our `Subject` object and implement the `ISubject` interface. Notice the `GetRequest()` method carries over in this object and has no changes:

```
// RealSubject
class Subject : MarshalByRefObject, ISubject
{
    public string GetRequest(){ return "Request Found"; }
}
```

Next we create the `Proxy`. The proxy for this example creates a new application domain and initializes the `ISubject` interface inside this new domain. For a more realistic implementation we could allow the proxy creation to supply the intended domain name and details for security and setting up the new domain code base:

```
// Remote Proxy Object
class SubjectProxy : ISubject
{
    private ISubject _subject;

    public SubjectProxy(
        string domainName,
        System.Security.Policy.Evidence evidence,
        System.AppDomainSetup appSetup)
    {
        AppDomain domain = System.AppDomain.CreateDomain(
            domainName, evidence, appSetup);
        //creates a {System.Runtime.Remoting.
                    Proxies.__TransparentProxy}
        //proxy stub to Examples.Proxy.Subject
        _subject = (ISubject)domain.CreateInstanceAndUnwrap(
            "Examples", "Examples.Proxy.Subject");
    }
```

Our proxy accesses the `GetRequest()` method on the `Subject` object and returns the result. Notice we are using the proxy stub to the `ISubject` interface created in the new application domain:

```
public string GetRequest()
{
    return _subject.GetRequest();
}
```

So if we wanted to call this proxy code in another domain, all we need to do is specify the domain details:

```
AppDomainSetup setup = new AppDomainSetup();
setup.ApplicationBase = path;
setup.ConfigurationFile = "(some file)";
```

And the security details that were needed to access our proxy:

```
// Set up the Evidence
Evidence baseEvidence = AppDomain.CurrentDomain.Evidence;
Evidence evidence = new Evidence(baseEvidence);
evidence.AddAssembly("(some assembly)");
evidence.AddHost("(some host)");
```

We pass these details into our proxy to identify the new domain, and call the `GetRequest()` method:

```
SubjectProxy proxy = new SubjectProxy("SubjectDomain",
                                      evidence, setup);
return proxy.GetRequest());
```

When we look at the code at run time, we can see where the new domain occurred that is accessing our proxy and see that our proxy passed the value from the `GetRequest()` method back into this domain:

```
using remoting.....System.Runtime.Remoting
Domain {SubjectDomain} created in:
C:\projects\DesignPatternsBook\src\Examples\bin\
Cross Domain Proxy results:Request Found
```

Comparison to Similar Patterns

Proxies give us another way to join code that exists in incompatible objects and domains. It works much like the Adapter

pattern, in that it hides the subject behind an interface and only allows controlled access to the objects that make up the subject. Bridges offer similar functionality. The Facade pattern is more like the proxy than these other two in that it hides a group or subsystem of objects. The difference in the Proxy pattern is it is designed to work with a little more separation of platforms, and handles object creation, security, and cleanup in a separate domain context.

What We Have Learned

Proxies allow us to maintain true separation of code from a point of view of ownership and control of underlying classes that the proxy provides access to. All access that a proxy controls should be handled by the proxy as a separate domain; in other words, the process flow model performs its operations separately from the executing call.

Related Patterns

- Adapter pattern
- Bridge pattern
- Facade pattern