

Design Patterns In C#

JEAN PAUL V.A

About the Author

Jean Paul V.A is a Software Developer working on Microsoft Technologies for the past 10 years. He has been passionate about programming and mentored lots of developers on .Net and related Technologies.

He has been holding MCPD and MCTS Certifications in:

- Windows Applications
- ASP.NET
- SQL Server
- SharePoint
- WCF

In the free time he would be focusing on writing Chapters and participating in Technology Forums including c-sharpcorner.com, MSDN Forums, dotnetpark.com. He has been the Member of Month holder for sites above and won the MindCracker MVP Award for 2011.

In the academics, he holds a Bachelor's Degree in Computer Science and Masters in Business Administration.

Presently he is working as Freelance Consultant over his native in India. His primary services include Development, Providing Architectural Decisions, Training Developers etc. He is presently running the blog: <http://jeanpaulva.com/>.

Apart from Programming he loves music and researching on stocks.

Dedications

This book is the compilation of my last 1 year learning effort in Design Patterns. It was a very interesting journey and I gained valuable assets towards my skill sets.

I would like to thank for all the encouragements I received from my friends as well as friends over c-sharpcorner.com.

Copyright © 2012 by Jean Paul V.A (www.jeanpaulva.com)

Contents

1. SINGLETON PATTERN	6
2. COMMAND PATTERN.....	9
3. ADAPTER PATTERN.....	15
4. DECORATOR PATTERN.....	18
5. STRATEGY PATTERN.....	22
6. TEMPLATE METHOD PATTERN.....	25
7. VISITOR PATTERN	30
8. OBSERVER PATTERN.....	34
9. BUILDER PATTERN.....	38
10. CHAIN OF RESPONSIBILITY PATTERN	41
11. ABSTRACT FACTORY PATTERN.....	46
12. FACTORY METHOD PATTERN	49
13. FLYWEIGHT PATTERN.....	53
14. PROXY PATTERN.....	57
15. FACADE PATTERN.....	61
16. STATE PATTERN	63
17. ITERATOR PATTERN.....	66
18. MEDIATOR PATTERN.....	70
19. MEMENTO PATTERN.....	74
20. PROTOTYPE PATTERN	78
21. BRIDGE PATTERN.....	82

22. INTERPRETER PATTERN	85
23. COMPOSITE PATTERN	90
REFERENCES.....	93
SOURCE CODE	93
CONCLUSION	94

1. Singleton Pattern

This is most popular pattern among the 23 Design Patterns. This pattern talks about efficient instance management.

Challenge

You are working on an application which requires Logging information into a file named **Application.Log**. The logging is managed by a class named **LogManager**. The LogManager through its constructor keeps a lock on the File.

The LogManager instance can be created throughout the application and it will result in Exceptions. Above that there is only instance requirement for the LogManager class.

The above problem can be solved using Singleton Pattern.

Definition

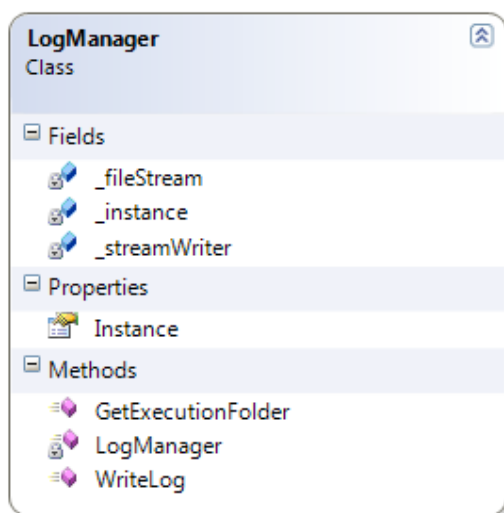
GoF Definition: "Ensure a class only has one instance, and provide a global point of access to it"

Implementation

To implement the specification above we have to perform the following:

- Make the Constructor Private and Create only 1 Instance
- Create a Static Property to Access the Instance

Following is the modified class definition according to Singleton Pattern.



Following is the code contained in LogManager class.

```
public class LogManager
{
    private static LogManager _instance;

    public static LogManager Instance
    {
        get
        {
            if (_instance == null)
                _instance = new LogManager();

            return _instance;
        }
    }

    private LogManager() // Constructor as Private
    {
        _fileStream = File.OpenWrite(GetExecutionFolder() +
"\Application.log");
        _streamWriter = new StreamWriter(_fileStream);
    }

    private FileStream _fileStream;
    private StreamWriter _streamWriter;

    public void WriteLog(string message)
    {
        StringBuilder formattedMessage = new StringBuilder();
        formattedMessage.AppendLine("Date: " +
DateTime.Now.ToString());
        formattedMessage.AppendLine("Message: " + message);

        _streamWriter.WriteLine(formattedMessage.ToString());
        _streamWriter.Flush();
    }

    public string GetExecutionFolder()
    {
        return
Path.GetDirectoryName(System.Reflection.Assembly.GetExecutingAssembly
().Location);
    }
}
```

You can see that the constructor is private here. As it is private outside classes cannot create instance of LogManager.

```
private LogManager() // Constructor as Private
{
    --
}
```

You can see the new Instance property which is static. It also ensures only one instance is created.

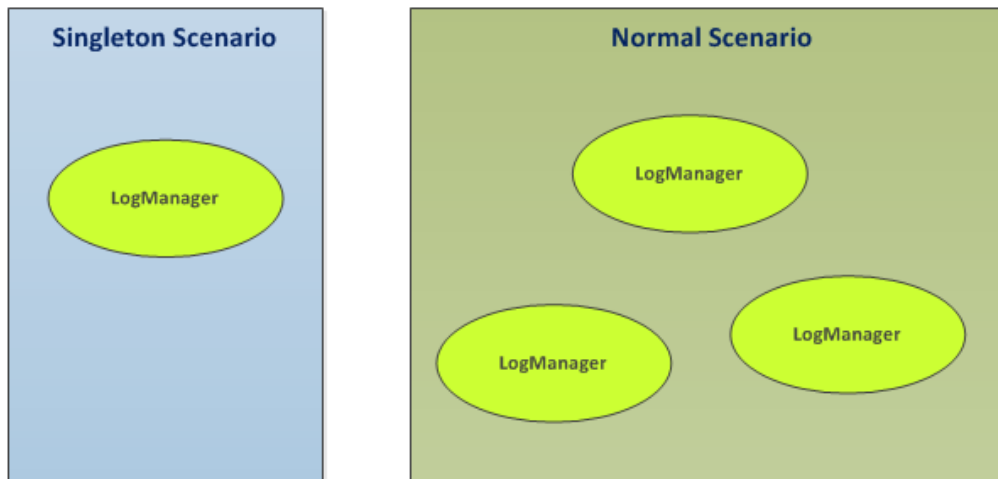
```
public static LogManager Instance
{
    get
    {
        if (_instance == null)
            _instance = new LogManager();

        return _instance;
    }
}
```

For invoking the write method we can use the following code:

```
LogManager.Instance.WriteLog("Test Writing");
```

Following image depicts the Number of Instances in Singleton and Normal Scenario.



The advantage of using Singleton is listed below:

- Instance Reuse makes Effective Memory Management
- File Locking Overheads are Avoided

The limitations of Singleton are listed below:

- Making Constructor as Private impose Restrictions
- More Overheads in Multi-threaded usage (But Thread Safe in .Net)

Summary

In this chapter we have seen what is Singleton pattern and a possible implementation of it using C#.NET. Multiton is another pattern which is often compared with Singleton.

2. Command Pattern

In this chapter I am trying to show the usage of Command Design Pattern. This is a popular pattern and it is making use of good object oriented constructs and simplifies the logic of enabling Undo functionality in our applications.

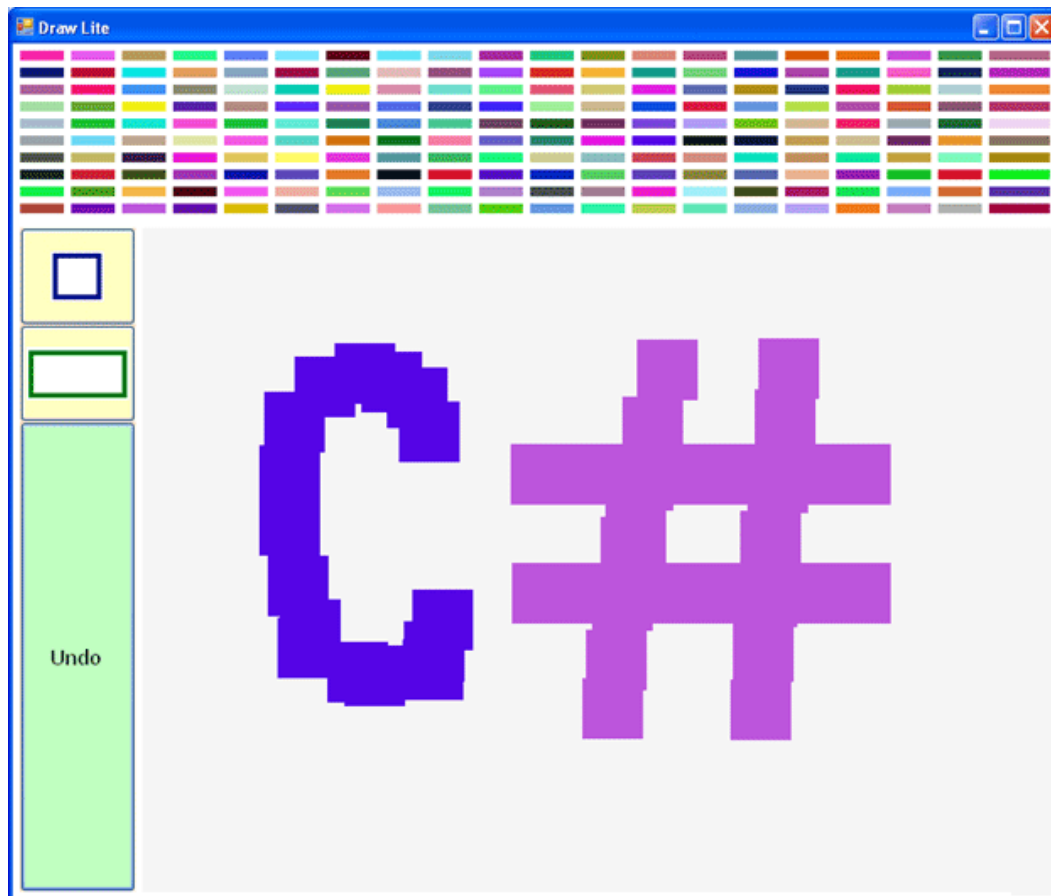
Challenge

You are working on a painting application. For each brush stroke you need to provide the Undo feature.

The brushes presently available are square and rectangle. In future more and more brushes will be available. Our challenge is to provide the user with Undo feature. Clearly, on clicking the undo button, the last stroke should be cleared from the canvas.

What would be your approach?

Screen Shot of Application



Definition

GoF Definition: "Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations"

Implementation

We can use the Command Pattern in the above scenario.

The pattern says that all the operations should be converted into objects, so that we log the objects. In future we can call Do() or Undo() operations on the object. The object should be smart enough to store the request parameters inside it to support the Do and Undo operations.

Command Pattern for our Scenario

Applied to our scenario, we have got two brushes:

- Square Brush
- Rectangle Brush

The other parameters would be Canvas which is the PictureBox control on which we draw and the X, Y location on mouse pointer on the canvas. Formulating this we can create 2 classes:

The SquareCommand class for square drawing operation and RectangleCommand class for rectangle operations.

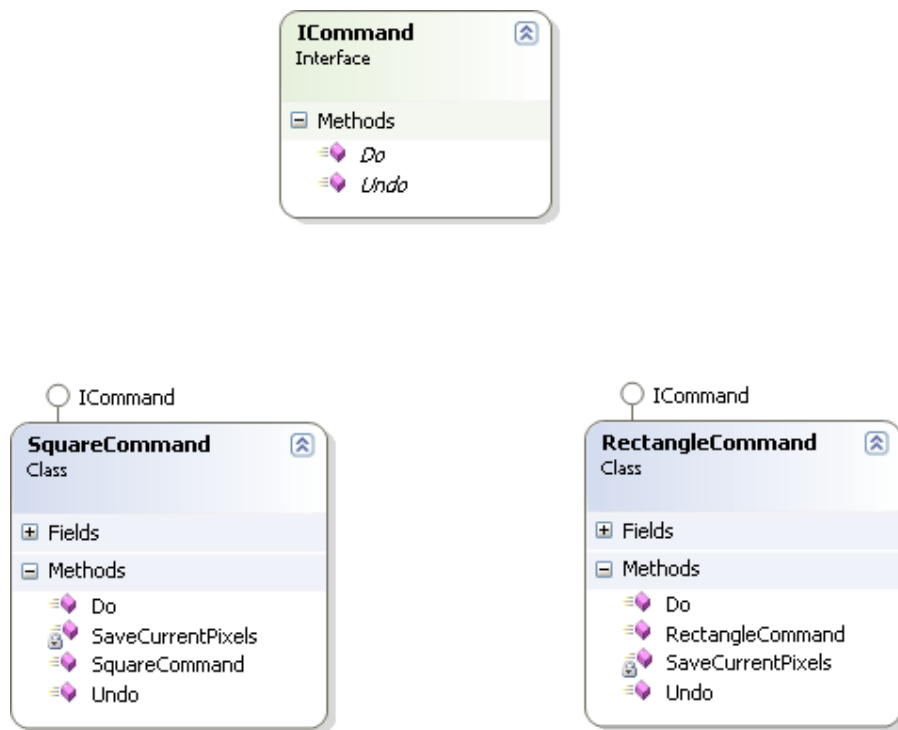
The Do() operation would be drawing a square for the square class and drawing a rectangle for the rectangle class.

We can formulate an interface called ICommand having the Do() and Undo() methods.

```
public interface ICommand
{
    void Do();
    void Undo();
}
```

The classes would be SquareCommand and RectangleCommand which implement s the above interface.

Class Diagram



When the application is executed, the user will be having a blank white screen, with a set of colors to choose from. The default brush selected will be Square and on clicking the canvas a new square will be drawn on the canvas. The default color selected is Blue and is stored in the variable `_activeColor`.

The form level variables declared are:

```

private ShapeEnum _activeShape = ShapeEnum.Square;
private Color _activeColor = Color.Blue;
private Bitmap _bitmap;
private Graphics _graphics;
  
```

For each mouse down operation a new square class will be created. That means if 10 clicks are made 10 square classes will be created and the `Do()` operation is called. An enumeration variable called `_activeShape` is used to keep track of the current shape selected by the user. The mouse down event would look like the following.

```

private void Canvas_MouseDown(object sender, MouseEventArgs e)
{
    // Each mouse down creates a new command class instance
    ICommand command = null;
    if (_activeShape == ShapeEnum.Square)
        command = new SquareCommand(_bitmap, _activeColor, e.X, e.Y);
    else if (_activeShape == ShapeEnum.Rectangle)
  
```

```

        command = new RectangleCommand(_bitmap, _activeColor, e.X, e.Y);
        command.Do();
        _commandStack.Push(command);
        RefreshUI();
    }

```

Let us take an example: assume the user clicked the mouse at Point X=100 and Y=200. As the `_activeShape` is Square, a new `SquareCommand` class instance is created with arguments `e.X` and `e.Y`. The remaining arguments are `bitmap` and the active color. Then the command instance would be pushed to a stack and the `Do()` operation is called.

The stack provides a storage for usage of the command instance later. The declaration of stack is:

```
private Stack<ICommand> _commandStack = new Stack<ICommand>();
```

Inside the SquareCommand Class

We can have a look on the Square class. Both the `SquareCommand` and `RectangleCommand` class will be having same implementations except in the `Do()` operation of drawing the respective shape.

```

public class SquareCommand : ICommand
{
    private Point _point;
    private Bitmap _bitmap;
    private Graphics _graphics;
    private Color _color;
    public SquareCommand(Bitmap bitmap, Color color, int x, int y)
    {
        _bitmap = bitmap;
        _graphics = Graphics.FromImage(_bitmap);
        _color = color;
        _point = new Point(x, y);
    }

    public void Do()
    {
        // Save the current pixel colors for a future UNDO perform
        SaveCurrentPixels();
        // Do the drawing
        _graphics.FillRectangle(new SolidBrush(_color),
            new Rectangle(_point.X, _point.Y, Width, Height));
    }
    private const int Width = 50;
    private const int Height = 50;
}

```

```

private IList<Color> _colors = new List<Color>();
private void SaveCurrentPixels()
{
    for (int i = _point.X; i < _point.X + Width; i++)
        for (int j = _point.Y; j < _point.Y + Height; j++)
            _colors.Add(_bitmap.GetPixel(i, j));
}
/// <summary>
/// Perform Undo by restoring back the pixels to previous colors
/// </summary>
public void Undo()
{
    int ix = 0;
    for (int i = _point.X; i < _point.X + Width; i++)
        for (int j = _point.Y; j < _point.Y + Height; j++)
            _bitmap.SetPixel(i, j, _colors[ix++]);
}
}

```

The constructor is called with the bitmap and the x, y positions which are then stored into class fields. When the Do() method is called the current pixel colors are saved into the _colors list. This would enable us to perform the Undo() method later. After that the graphics.FillRectangle() method is called to draw the square. The width and height of the square would be set to 50 pixels using the constants Width and Height respectively.

The Undo() method just restores the previous pixel values using the bitmap.SetPixel method() by iterating through each pixel starting from x, y position.

Performing the Undo() on clicking Undo button

As we encapsulated each operation to a class with parameters and logged them into the command stack, it is now easier to call the Undo() method. Please remind that on clicking the Undo button we have to get the last operation instance and call the Undo() method of it, and removing it from the stack.

The code for it is:

```

private void UndoButton_Click(object sender, EventArgs e)
{
    // Check command stack contains items
    if (_commandStack.Count > 0)
    {
        // Remove the last command
        ICommand lastCommand = _commandStack.Pop();
        // Call the Undo method
        lastCommand.Undo();
    }
}

```

```
}  
    RefreshUI();  
}
```

First, the command stack count is checked to ensure there are commands inside it. Then the last command is popped out using the Pop() method. It will give us the last instance in the stack as well as removes it. Then, the Undo() operation is invoked. After, the RefreshUI() method is called to update the UI with the changes.

Summary

The command pattern provides an object oriented manner to provide Undo feature to our application in a flexible manner.

The creation of object instances for each command is one of the drawback of this method which usually people say about. But the pattern provides an object oriented way for achieving Undo functionality.

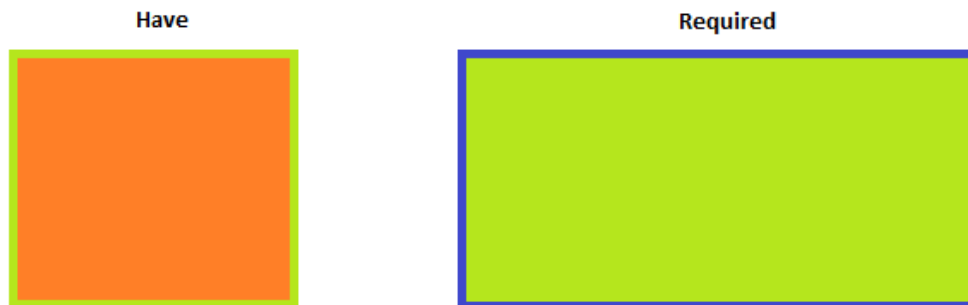
The attachment contains the application we have discussed.

3. Adapter Pattern

Most of you might have heard about Adapter Pattern. It is a pattern commonly used in our applications but without knowing it. Adapter Pattern is one among the 23 Design Patterns. In this chapter I would like to examine this pattern using a simple example.

Challenge

You are working on a Square class. You need to find the Area of it using Calculator class. But the Calculator class only takes Rectangle class as input. How to solve this scenario?



Definition

GoF Definition: "Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces."

Implementation

Following are the class definitions for Rectangle and Calculator.

```
public class Rectangle
{
    public int Width;
    public int Height;
}

public class Calculator
{
    public int GetArea(Rectangle rectangle)
    {
        int area = rectangle.Width * rectangle.Height;

        return area;
    }
}
```

As we can see from the above example an instance of Rectangle is needed to calculate the area. If we have a square class of definition below, the calculation cannot be done.

```
public class Square
{
    public int Size;
}
```

Here we have to create a new CalculatorAdapter to get the work done.

```
public class CalculatorAdapter
{
    public int GetArea(Square square)
    {
        Calculator calculator = new Calculator();

        Rectangle rectangle = new Rectangle();
        rectangle.Width = rectangle.Height = square.Size;

        int area = calculator.GetArea(rectangle);

        return area;
    }
}
```

The CalculatorAdapter performs the following functions:

- Takes the Square parameter
- Convert Square to Rectangle
- Call the original Calculator.GetArea() method
- Return the value received

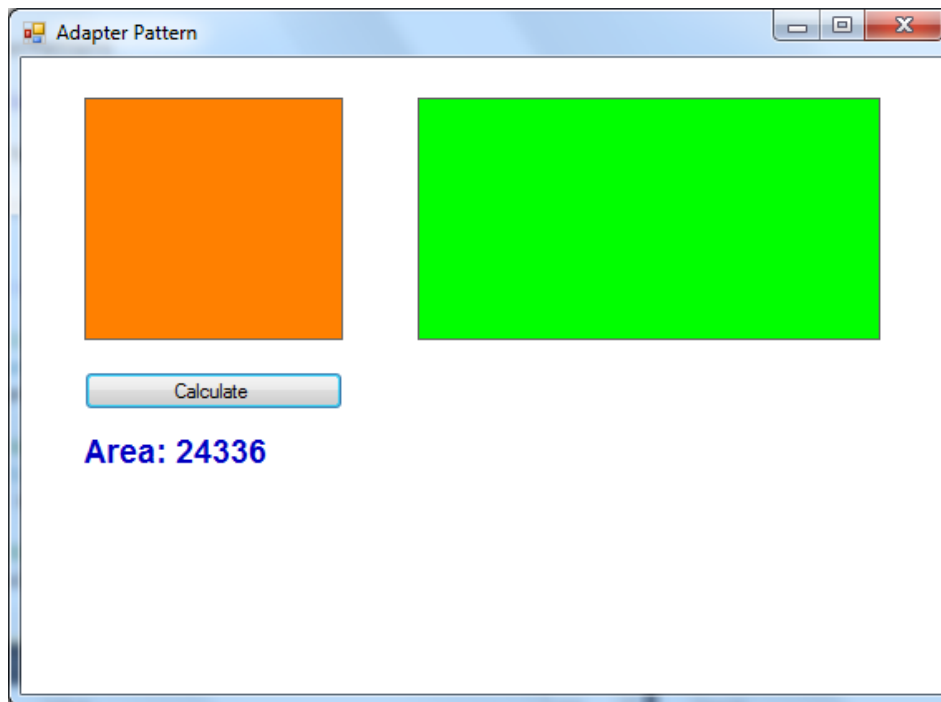
The invoking code is shown below:

```
// Create Square class and assign Size from UI
Square square = new Square();
square.Size = SquarePanel.Width;

// Use Adapter to calculate the area
CalculatorAdapter adapter = new CalculatorAdapter();
int area = adapter.GetArea(square);

// Display the result back to UI
ResultLabel.Text = "Area: " + area.ToString();
```

On running the sample application we can see the following results.

**Note**

We can have other examples including Interfaces to show the above pattern. For simplicity I have avoided the interfaces. In real life the AC to DC adapter is an example of the Adapter pattern as it makes the incompatible device and power supply work together.

Summary

In this chapter we have explored Adapter pattern. This pattern is useful in scenarios where incompatible types are dealt with. The associated source code contains the example application we have discussed.

4. Decorator Pattern

In this chapter I would like to demonstrate the Decorator Pattern. It provides a new way of adding responsibilities in the runtime and thus a good alternative to sub classing.

Challenge

You have an Album class today which is just blank now. Tomorrow the customer wants a Christmas tree on that. You need to achieve this without modifying the Album class.

How to achieve this during runtime?

Definition

GoF Definition: "Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality"

Closed for Modification and Open for Extension

One of the main challenges we face in development is Change. The Closed for Modification and Open for Extension principle says a new functionality can be added by keeping the original code unchanged.

Example:

We will modify the original Album class to incorporate the Christmas tree on it. This should not be the best approach. There is a better approach to this. We can still keep the Album class unchanged and add the Christmas tree to it. Everything happens in the runtime – that is the cool part of it.

Some more examples

We can see real life controls like Form, Button etc. There would be a Form class with built-in functionality. Still the user can use it and add new controls to it / extend the functionality. Here we will be basically deriving from the existing Form/Button class and add new methods or properties to it.

The difference between the above approach and Decorator pattern is that, in the Decorator pattern, it is done during runtime.

Conclusion on Change

So basically we can conclude that whenever changes are required, the possible solutions could be:

- Change the original class
- Subclass it and create instance of subclass
- Use Decorator Pattern and still using the original class instance

Here we are going to see how we can use Decorator Pattern to help with the following scenario.

Requirement

The requirement here would be to provide a default Album object and based on dynamic requirement from the user in runtime, we have to draw other pictures to the album.



Design

Our first class would be the Album class which has a Graphics object as parameter.

It contains a Draw() method which is virtual and just clears the graphics object.

```
public class Album
{
    public Graphics Graphics
    {
        get;
        set;
    }
}
```

```

public Album()
{
}

public Album(Graphics graphics)
{
    Graphics = graphics;
}

public virtual void Draw()
{
    Graphics.Clear(Color.White);
}
}

```

Decorator

We are adding the class named AlbumDecorator which will serve as the base class for all decorators.

```

public abstract class AlbumDecorator : Album
{
    protected Album _album;

    public AlbumDecorator(Album album)
    {
        _album = album;
    }

    public override void Draw()
    {
        _album.Draw();
    }
}

```

It takes an Album class as parameter in the constructor.

There are ChristmasTreeDecorator, SantaClausDecorator, StarDecorator deriving from AlbumDecorator:

```

public class ChristmasTreeDecorator : AlbumDecorator
public class SantaClausDecorator : AlbumDecorator
public class StarDecorator : AlbumDecorator

```

Each class deriving from AlbumDecorator has it's own picture to draw.

Invoking

In the main form we create an instance of Album class and assign it to form field `_album`.

```
private Album _album;  
_album = new Album(_graphics);  
_album.Draw();
```

In the runtime, when user wants a Christmas Tree, an instance of `ChristmasTreeDecorator` is created.

```
_album = new ChristmasTreeDecorator(_album);  
_album.Draw();
```

In the above code we can see the same `_album.Draw()` method is called.

How it works

Whenever we call the `Draw()` method of a decorator class, it in turns calls the original `Album.Draw()`. After that it will call it's own `Draw()` method. In this way we can pass the same album instance to multiple decorators. If there are 10 decorators, all the decorator `Draw()` methods will be invoked.

You can test this by placing a breakpoint inside the `StarDecorator Draw()` method.

Note

In the real world scenario we can use Decorator Pattern to calculate Taxes as decorators for the Item Price. The Tax, Extra Tax etc. can be wrapped as decorators around the original item price.

Summary

Using decorator we can add dynamic responsibilities to an object in runtime. This provides us the flexibility of creating an instance of decorators on an as-needed basis. This would provide a real advantage in scenarios where the additional responsibility increases the use of memory. The attachment contains the example we have discussed.

5. Strategy Pattern

In this chapter I would like to explore the flexibility of Strategy design pattern.

Challenge

You are working on a Gaming application. The player can select between two different guns:

1. Pistol having 5 bullets
2. Grenade Launcher having 20 bullets

In the normal coding scenario, if the gamer changes the gun we have to do lot of if conditions in the fire method, in the draw gun method, in the bullet animation methods.

How to do a better approach?

Definition

GoF Definition: "Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it"

Implementation

We can avoid the above complications by switching to the Strategy pattern. Our solution would be: Extract the difference implementations into appropriate classes and switch between the classes.

Code Explained

The core classes involved are:

- Gamer
- PistolGun implementing IGun
- GrenadeLauncher implementing IGun

Whenever the user changes the gun the Gamer.Gun property is switched between PistolGun and GrenadeLauncher instances.

Here the IGun interface extracts the part which differs for different guns.

```

public interface IGun
{
    void Fire();

    void Draw();

    int Bullets { get; set; }

    PictureBox GunBox { get; set; }

    PictureBox BulletBox { get; set; }
}

```

The Gamer class which maintains the Gun and refreshes the Picture Box would be looking like:

```

public class Gamer
{
    private Label _bulletsLabel;

    public Gamer(Label bulletsLabel)
    {
        _bulletsLabel = bulletsLabel;
    }

    private IGun _gun;

    public IGun Gun
    {
        get { return _gun; }
        set { _gun = value; RefreshGunInfo(); }
    }

    private void RefreshGunInfo()
    {
        _gun.Draw();
        ShowBulletsInfo();
    }

    private void ShowBulletsInfo()
    {
        _bulletsLabel.Text = "Bullets: " + _gun.Bullets.ToString();
    }

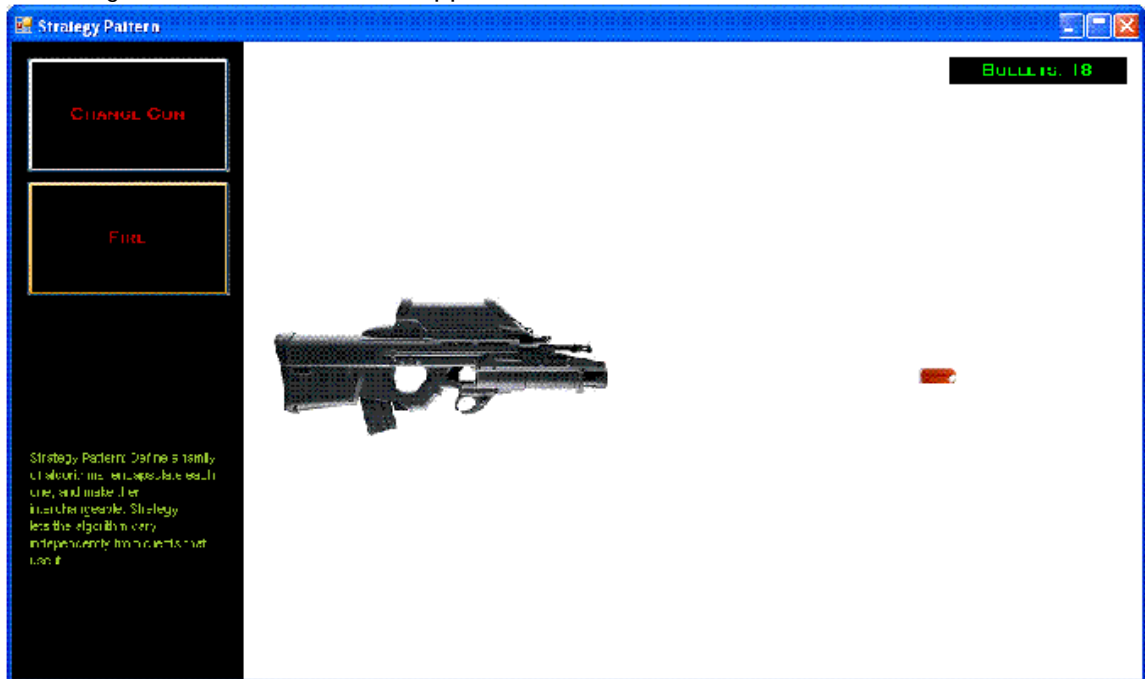
    public void Fire()
    {
        _gun.Fire();

        ShowBulletsInfo();
    }
}

```

Screenshot of Application

Following is the screenshot of the application:



Summary

Strategy pattern helps us to make the code more cleaner, highly object oriented and easier to manage and extensible. The attachment contains the example application we have discussed.

6. Template Method Pattern

Template Method is a widely used design pattern and it provides great flexibility in our designs. This is one of my favourite patterns.

Challenge

You are creating machines for creating Pizza and Burger. When we look closely we can see all the machines have some operations in common and in the same order.

- Start
- Produce
- Stop

The Start operation consists of turning on the machines, do the system check for any troubles and turning on the indicators.

The Produce operation does the respective production of item.

The Stop operation shutdowns the internal workings of the machine, turns off the indicators and power offs the machine.

The challenge is to create more machines like Cheese Burger, Pan Pizza with less custom implementations. We need to take care that the duplicate codes are avoided in our design.

Definition

GoF Definition: "Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure"

Implementation

We can see that all the operation appears in the same sequence. Often the Start and Stop operation are common to both Pizza and Burger machines.

So we can define a skeleton of algorithm (or the order of invocation) like below:

1. Start
2. Produce
3. Stop

We can see all the machines have this order of execution. Only the Produce operation will be different.

So creating an Execute() method to incorporate this would look like:

```
public class Machine
{
    public void Execute()
    {
        Start();
        Produce();
        Stop();
    }
}
```

By calling the Execute() method we can invoke all the 3 methods in the right order. Keeping the Execute() method in the base class as public, we can create Produce() method as virtual so that it can be overridden. For more customization we are making all the 3 methods as virtual.

Note

For virtual methods, a default implementation will be provided in the base class. The derived class may/may not override it.

For abstract methods, we need to override them in the derived class. There won't be any default implementation in the base class.

The modified Machine class will look like below:

```
public class Machine
{
    public void Execute()
    {
        Start();
        Produce();
        Stop();
    }

    protected virtual void Start()
    {
        Trace.WriteLine("Machine.Starting..");
    }

    protected virtual void Produce()
    {
        Trace.WriteLine("Machine.Producing..");
    }

    protected virtual void Stop()
    {
        Trace.WriteLine("Machine.Stopping..");
    }
}
```

Please note the class uses **virtual** keyword so that the derived classes can override.

The derived class PizzaMachine will look like:

```
public class PizzaMachine : Machine
{
    protected override void Produce()
    {
        Trace.WriteLine( "PizzaMachine.Producing.." );
    }
}
```

The derived class BurgerMachine will look like:

```
public class BurgerMachine : Machine
{
    protected override void Produce()
    {
        Trace.WriteLine( "BurgerMachine.Producing.." );
    }
}
```

Please note that both of them derives from **Machine** class and use the keyword **override** to custom implement the Produce() method.

Execution

Now we can execute the PizzaMachine using the Execute() method of base class.

```
new PizzaMachine().Execute();
```

Let us examine how the order of execution works.

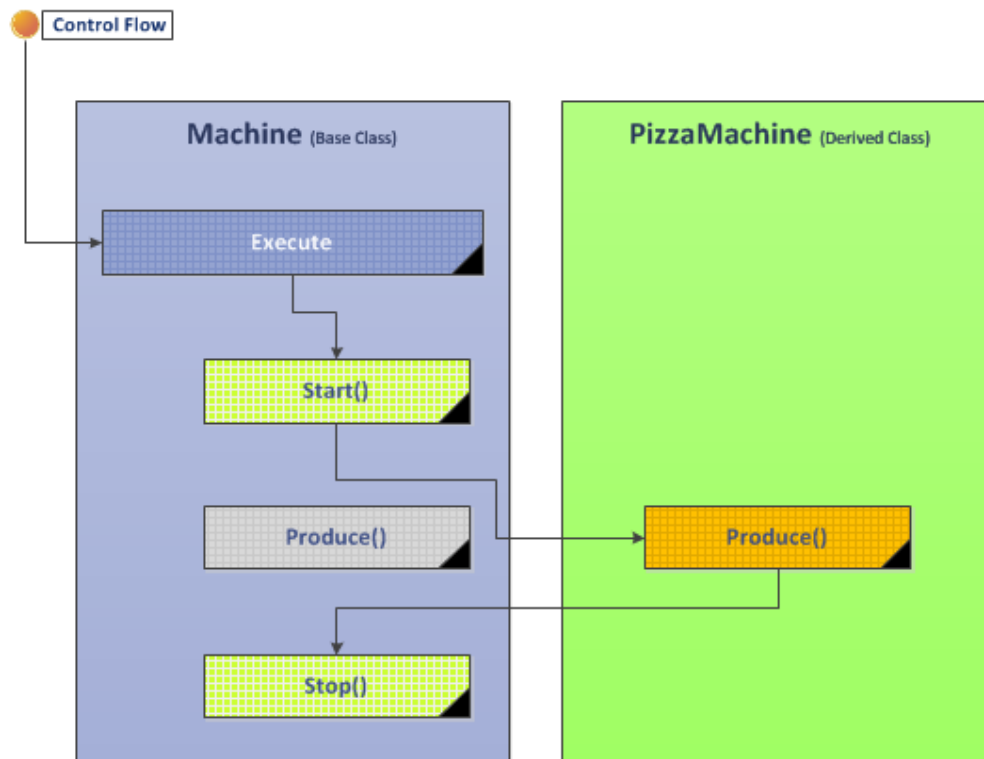
1. Machine.Execute() is invoked
2. Machine.Start() is invoked
3. PizzaMachine.Produce() is invoked
4. Machine.Stop() is invoked

Please note the step 3 above, in which the execution shifts from Machine class to PizzaMachine class. This is the power of Template Method pattern. Here the order of execution is maintained and a custom production is done.

Note

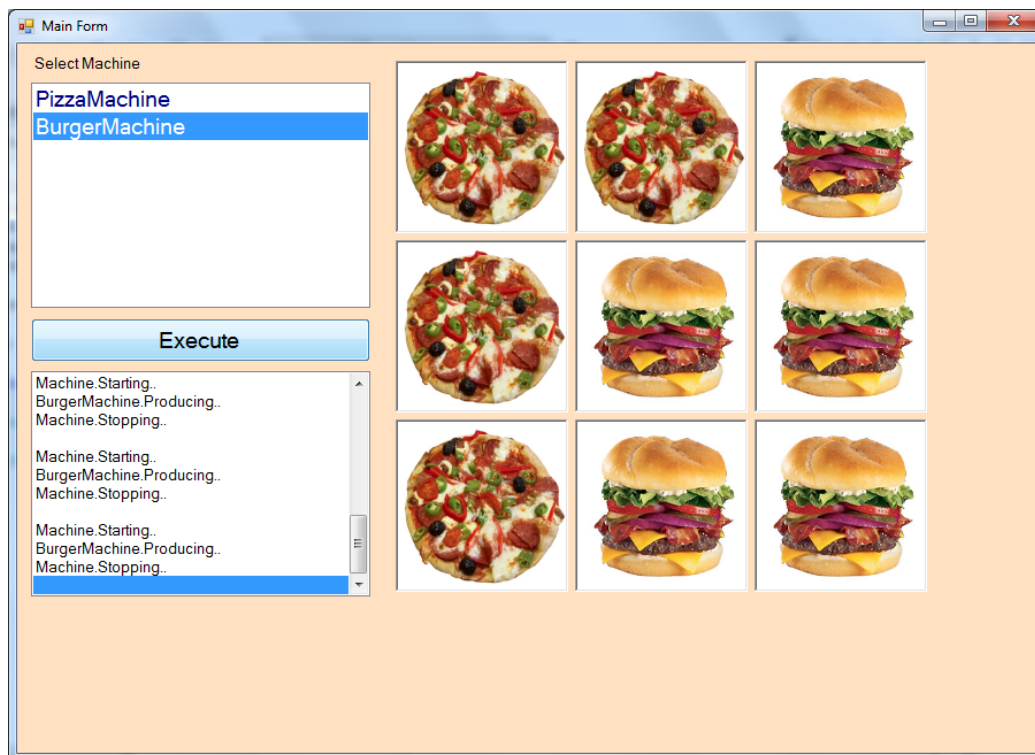
The underlying detail of how the above code works goes to Virtual Method Table (VMT) where the base class method address is kept in a table. The derived classes will be having a new address for the derived method. This will be replacing the original base class method address.

The control flow is depicted in the image below.



Test Application

A test application is created with a GUI using WinForms.



Summary

In this chapter we have seen the advantage of Template Method design pattern and the problem it tries to address. The source code contains the classes and application explained.

7. Visitor Pattern

Visitor Pattern is one among the 23 design patterns. It is less known but holds a good idea for OOPs enthusiasts.

Challenge

You are having a list of objects with the following class structure:

```
public class Member
{
    public string Name;
    public DateTime DateOfBirth;
}
```

You need to do an operation of selecting all Member having age greater than 18.

One way of the solution is add a new property called `IsAboveEighteen` and set the value by iterating over the items and comparing with current date. But this requires more processing.

We can achieve the same using Visitor pattern by adding a property which operations on the existing properties and returns the value.

Definition

GoF Definition: "Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates"

Implementation

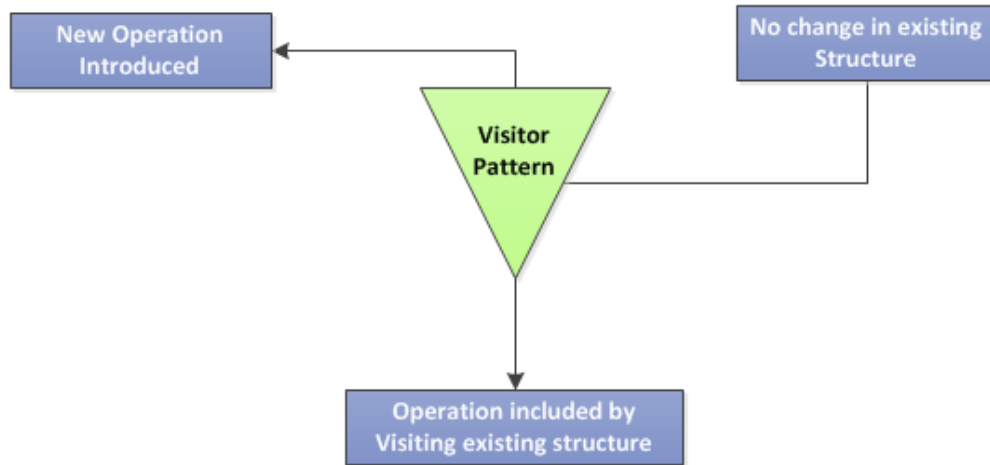
Introducing a new property named **IsAboveEighteen** and in the getter implement the code to read `DateOfBirth` property to calculate the value.

```
public class Member
{
    public string Name;
    public DateTime DateOfBirth;

    public bool IsAboveEighteen
    {
        get
        {
            bool result = (DateTime.Now - this.DateOfBirth).TotalDays > 365 * 18;

            return result;
        }
    }
}
```

In the above example, the new property explores the existing property values to calculate its own value. The advantage is that there is no change of structure and no extra operations to achieve the desired result.



The following unit tests operate on the new property and display the result having `IsAboveEighteen` as true.

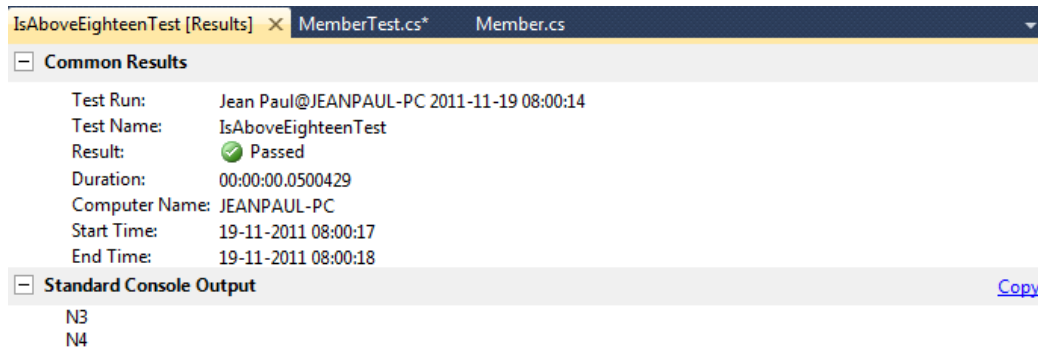
```

[TestMethod()]
public void IsAboveEighteenTest()
{
    IList<Member> list = new List<Member>()
    {
        new Member() { Name = "N1", DateOfBirth= new DateTime(2000, 1, 1)},
        new Member() { Name = "N2", DateOfBirth= new DateTime(2000, 1, 1)},
        new Member() { Name = "N3", DateOfBirth= new DateTime(1990, 1, 1)},
        new Member() { Name = "N4", DateOfBirth= new DateTime(1980, 1, 1)}
    };

    var selectedList = list.Where(m => m.IsAboveEighteen);
    foreach (Member member in selectedList)
        Console.WriteLine(member.Name);

    Assert.AreEqual(2, selectedList.Count());
}
  
```

On running the test we can see the following output.



Extending Visitor in Sql Server

In Sql Server the same functionality can be achieved using Computed Columns. Even though Visitor Pattern is an Object Oriented Extension we can use the concept in database too.

Let us explore this with a simple example. We are having a table to store Transaction having Quantity and Price. The table is populated with data. We need to get the TotalPrice which is Quantity multiplied by Price. Without doing any data updating we can use Computed Column as shown below to achieve the results.

Table - dbo.Transaction				
	Id	Item	Quantity	Price
▶	1	Item1	10	5
	2	Item2	10	8
*	NULL	NULL	NULL	NULL

Table - dbo.Transaction		
Column Name	Data Type	Allow Nulls
Id	int	<input type="checkbox"/>
Item	varchar(50)	<input checked="" type="checkbox"/>
Quantity	float	<input checked="" type="checkbox"/>
Price	float	<input checked="" type="checkbox"/>
TotalPrice		<input checked="" type="checkbox"/>

Column Properties	
(General)	
(Name)	TotalPrice
Allow Nulls	Yes
Data Type	
Default Value or Binding	
Length	
Table Designer	
Collation	<database default>
Computed Column Specification	Quantity * Price
(Formula)	Quantity * Price

The result is shown below:

Table - dbo.Transaction*		Table - dbo.Transaction			
	Id	Item	Quantity	Price	TotalPrice
▶	1	Item1	10	5	50
	2	Item2	10	8	80
*	NULL	NULL	NULL	NULL	NULL

The above result is achieved without doing any data updating.

Summary

In this chapter we have explored the Visitor pattern. It allows us to be add more functionality without doing much change in the structure. Keeping this pattern in mind often helps in a better way of architecting.

8. Observer Pattern

In this chapter we can discuss about the Observer design pattern. This pattern is a widely used one.

Challenge

You are working on an application where there is a color to be selected by the user. Based on the color selection n number of forms need to be notified. One way of achieving this is iterate through all the forms available and notify them. This results in unwanted iterations and notifications.

How to provide a better solution?

Definition

GoF Definition: "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically"

Implementation

We can use the Observer pattern for solving the above problem. Subject and Observers are the key terms of the pattern. Here Color acts as the subject.

The Observer pattern is to notify the interested observers about some change occurred. We can add more observers in runtime as well as remove them.

Example: We have a form to select the color. For each color change we need to update the entire application. There will be observers listening to the color change event for updating themselves.

Subject and Observers

The two important key terms in the pattern are Subject and Observer.

Subject is the object which holds the value and takes responsibility in notifying the observers when the value is changed. The subject could be a database change, property change or so.

We can conclude that the subject contains the following method implementations.

```
public interface ISubject
{
    void Register(IObserver observer);
    void Unregister(IObserver observer);

    void Notify();
}
```

The Observer is the object listening to the subject's change. Basically it will be having its own updating/calculating routine that runs when get notified.

```
public interface IObservable
{
    void ColorChanged(Color newColor);
}
```

In the above example, we are using an observer interface which has a ColorChanged method. So the interested observers should implement this interface to get notified.

There will be only one Subject and multiple number of Observers.

Registering and Unregistering

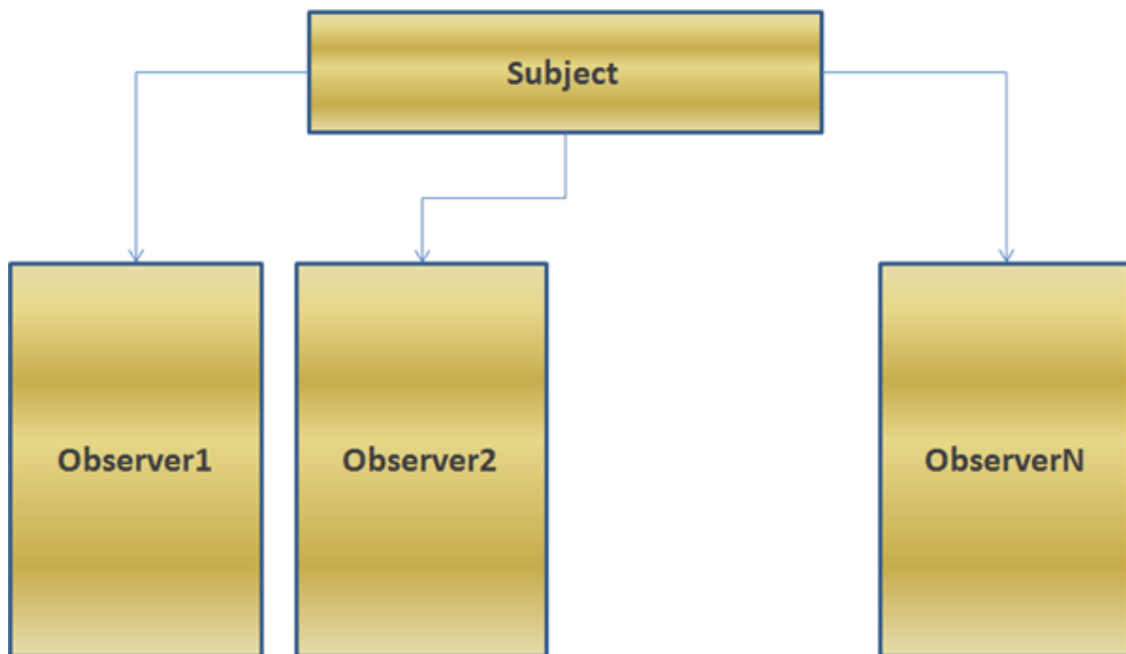
In the above interface, the observer can use the Register() method to get notified about changes. Anytime, it can unregister about notifications using the Unregister() method.

Notifying

The Notify() method will take care of calling the listening observers.

Associations

The Subject and Observer objects will be having a one-to-many association.



Using the Code

The associated code is having a main form, where the Subject would be a class named ColorSubject.

```

public class ColorSubject : ISubject
{
    private Color _Color = Color.Blue;

    public Color Color
    {
        get { return _Color; }
        set
        {
            _Color = value;
            Notify();
        }
    }

    #region ISubject Members

    private HashSet<IObserver> _observers = new HashSet<IObserver>();

    public void Register(IObserver observer)
    {
        _observers.Add(observer);
    }

    public void Unregister(IObserver observer)
    {
        _observers.Remove(observer);
    }

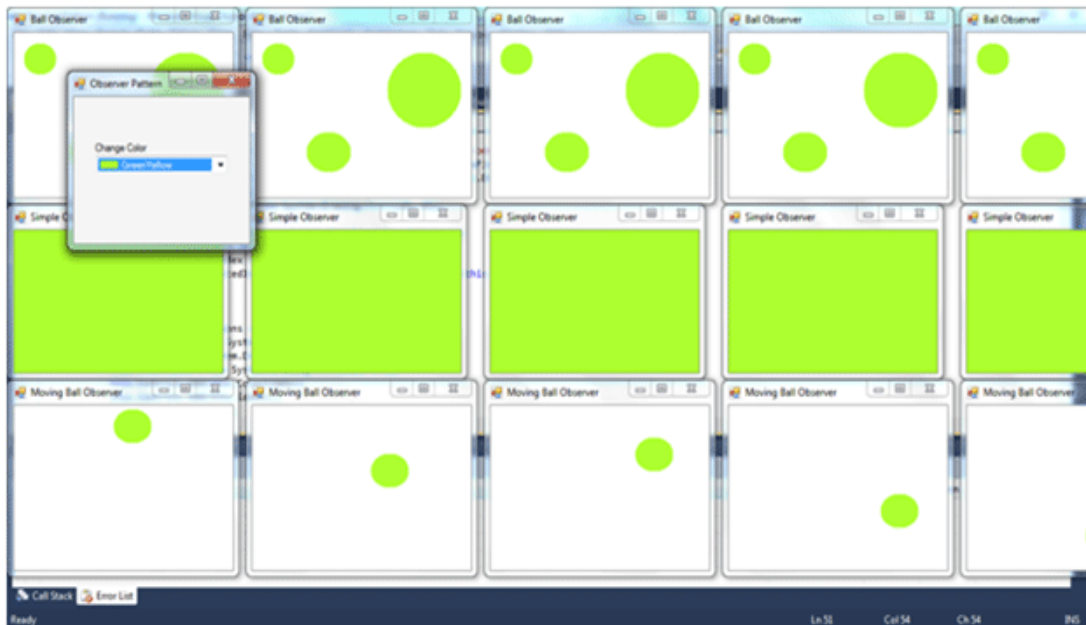
    public void Notify()
    {
        _observers.ToList().ForEach(o => o.ColorChanged(Color));
    }

    #endregion
}

```

The class implements ISubject interface and thus takes care of registering, unregistering and notifying the interested observers. All the observers keep registered with the ColorSubject object.

Screen Shot of Application



On running the associated project, we can see a color selector form, which acts as the subject. All the other forms are observers listening to the ColorChanged event.

Note

The multicast event model in .Net can also be considered as an observer pattern. Here the interested parties register a method with the subject (might be a button) and whenever the button is clicked (an event) it invokes the registered observers (subscribers)

Summary

In this chapter we have seen what Observer pattern is and an example implementation. The attachment contains the source code of the application we have discussed.

9. Builder Pattern

In this chapter I am trying to demonstrate the usage of Builder Pattern using a Computer Assembling Robot.

Challenge

In a computer manufacturing unit, depending on the configuration they have to create computers. The challenge is that the computer peripheral is chosen in the runtime. The order can contain n number of computers based on the same configuration. This could be the right example to implement the Builder pattern where each robot will be setup with a particular computer configuration. The robot will continue creating the same configuration based computer n times.

Computer Peripherals

Processor: Intel / AMD

Monitor: Samsung / LG

Speakers: None / Stereo / Surround

How to provide a better solution that allows to choose the peripherals at runtime?

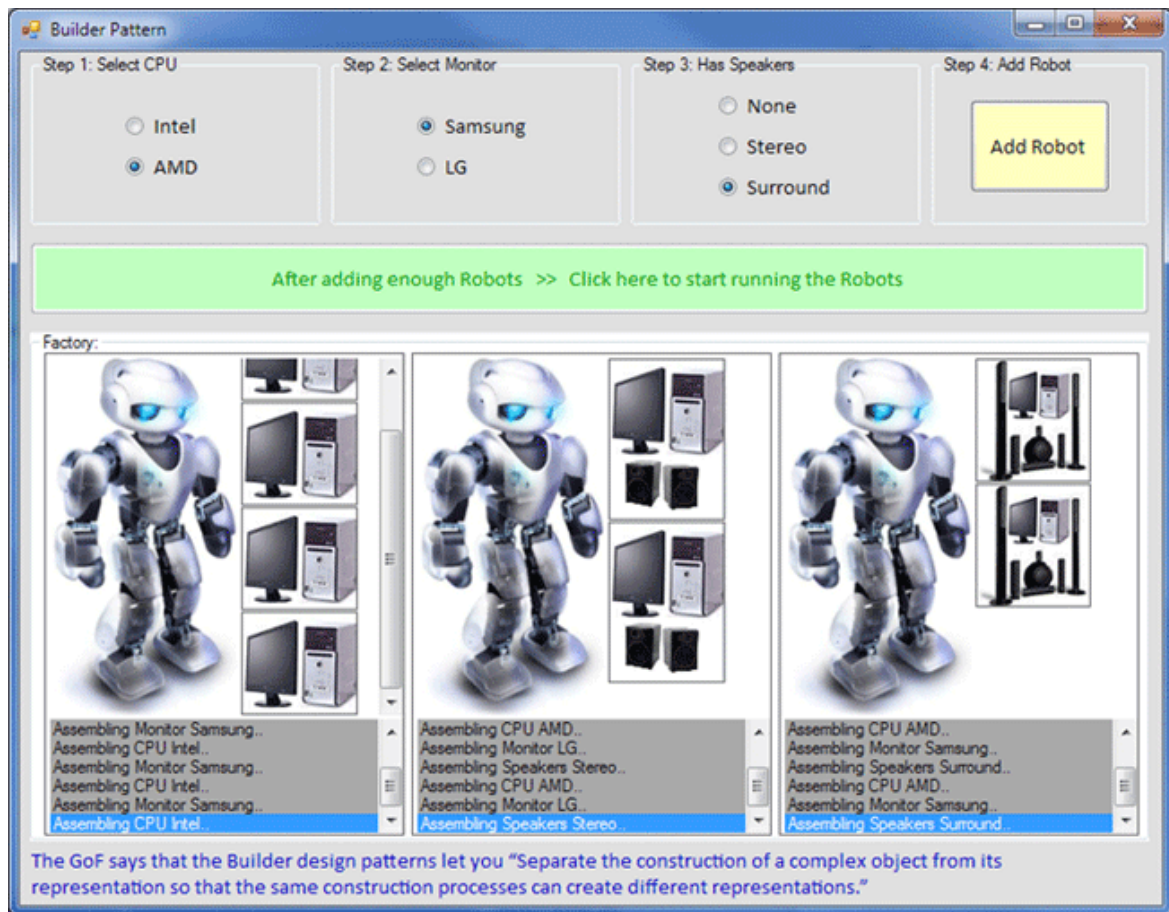
Definition

GoF Definition: "Separate the construction of a complex object from its representation so that the same construction processes can create different representations"

Implementation

We can use the Builder Pattern to address the above problem. The pattern allows predefined processes with the steps configurable in the runtime.

The application contains a Windows Form which allows the user to choose the configuration. The virtual factory provides up to 3 robots to be added before the actual assembling takes place.



Code Explained

Following is the screen shot of the main class named Robot. The class contains a property called Peripherals which can be used to configure the peripherals added: like CPU as Intel, Monitor as Samsung etc.

After adding the peripherals, we can call the Create() method. It will iterate through the Peripherals dictionary and calls the appropriate methods to integrate the peripherals. This allows us to choose the peripherals at run time as well as the sequence too.

If the Peripheral dictionary contains CPU then the AssembleCPU() method is invoked.

If the Peripheral dictionary contains Monitor then the AssembleMonitor () method is invoked.

```

public partial class Robot : UserControl
{
    /// <summary>
    /// Dictionary containing peripheral and the option
    /// </summary>
    public Dictionary<Peripheral, string> Peripherals = new Dictionary<Peripheral, string>

    /// <summary>
    /// The core method used to create different representations
    /// </summary>
    public void Create()
    {
        while (true)
        {
            // Create new Computer
            _computer = new Bitmap(Images.Computer);

            foreach (Peripheral peripheral in Peripherals.Keys)
            {
                if (peripheral == Peripheral.Processor)
                    AssembleCPU(Peripherals[peripheral]);

                if (peripheral == Peripheral.Monitor)
                    AssembleMonitor(Peripherals[peripheral]);

                if (peripheral == Peripheral.Speakers)
                    AssembleSpeakers(Peripherals[peripheral]);
            }

            AddToStore();
        }
    }
}

```

The class Peripheral is an enumeration as following:

```

public enum Peripheral
{
    Processor,
    Monitor,
    Speakers
}

```

Summary

In this chapter we have seen the usage of Builder pattern. The attachment contains the source code of example we have discussed.

10. Chain of Responsibility Pattern

Chain of Responsibility is one among the 23 Design Patterns by Gang of Four. It is an interesting pattern and similar to Observer pattern. In this chapter we are exploring the Chain of Responsibility pattern.

I am using the same Challenge and Solution style for explaining this pattern.

Challenge

You are working on an application in which the **Logic** class generates various messages. The messages are of two types.

- Normal Priority
- High Priority

The Normal Priority messages are to be processed by **Logger** class and High Priority messages by **Emailer** class.

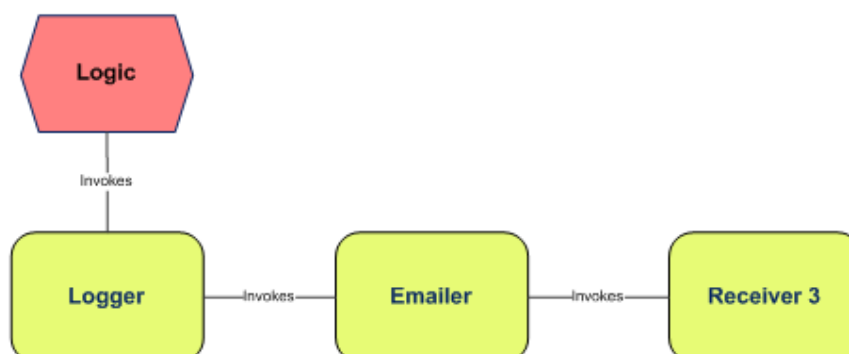
You have to make the design in such a way that the **Logic** class need not think about right handler of the message. It will just send the message.

How the design will proceed?

Definition

GoF Definition: "Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it"

Control Flow



Implementation

Following is the definition for Logic class:

```
public class Logic
{
    public IReceiver Receiver;

    public void CreateMessage(Message message)
    {
        if (Receiver != null)
            Receiver.HandleMessage(message);
    }
}
```

We can see from the above code that the Logic class has a property of type IReceiver. On the CreateMessage method this Receiver is used to handle the message. So only one receiver is registered by the Logic class instance. Following are the definition of Message class and MessagePriority enumeration:

```
public class Message
{
    public string Text;
    public MessagePriority Priority;
}

public enum MessagePriority
{
    Normal,
    High
}
```

Following are the Receiver Interface and Implementation classes:

```
public interface IReceiver
{
    bool HandleMessage(Message message);
}

public class Logger : IReceiver
{
    private IReceiver _nextReceiver;

    public Logger(IReceiver nextReceiver)
    {
        _nextReceiver = nextReceiver;
    }

    public bool HandleMessage(Message message)
    {
        if (message.Priority == MessagePriority.Normal)
        {
            Trace.WriteLine(message.Text + " : Logger processed it!");
            return true;
        }
        else
        {
            if (_nextReceiver != null)
                _nextReceiver.HandleMessage(message);
        }
    }
}
```

```

        return false;
    }
}

public class Emailer : IReceiver
{
    private IReceiver _nextReceiver;

    public Emailer(IReceiver nextReceiver)
    {
        _nextReceiver = nextReceiver;
    }

    public bool HandleMessage(Message message)
    {
        if (message.Priority == MessagePriority.High)
        {
            Trace.WriteLine(message.Text + " : Emailer processed it!");
            return true;
        }
        else
        {
            if (_nextReceiver != null)
                _nextReceiver.HandleMessage(message);
        }

        return false;
    }
}

```

From the above code we can see that each receiver class checks the message priority and processes it. If the priority is not matching it is send to the next receiver in the chain.

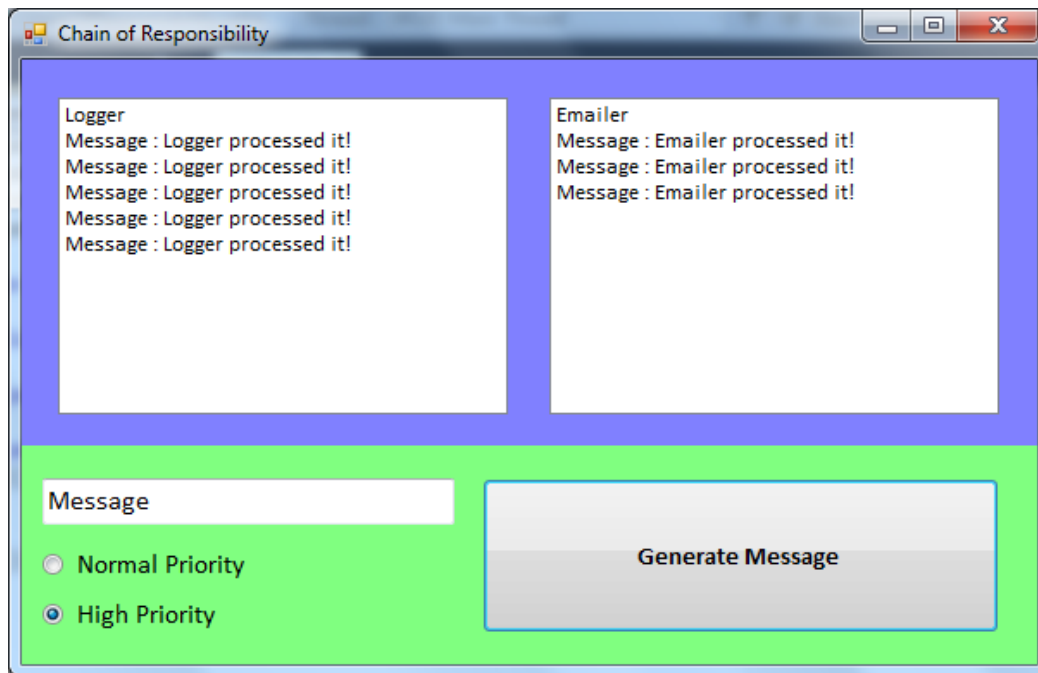
Here each receiver takes care of passing the unprocessed message. The next receiver is stored through the constructor of each receiver.

The chain will be executed until:

- Message is processed
- Receiver chain exhausted

Code Execution

The windows forms application attached can be used to test the control flow.



Here the chain is built as following:

- Logic class holds first receiver which is Logger
- Logger class holds the next receiver which is Emailer
- Emailer class have null receiver denoting end of chain

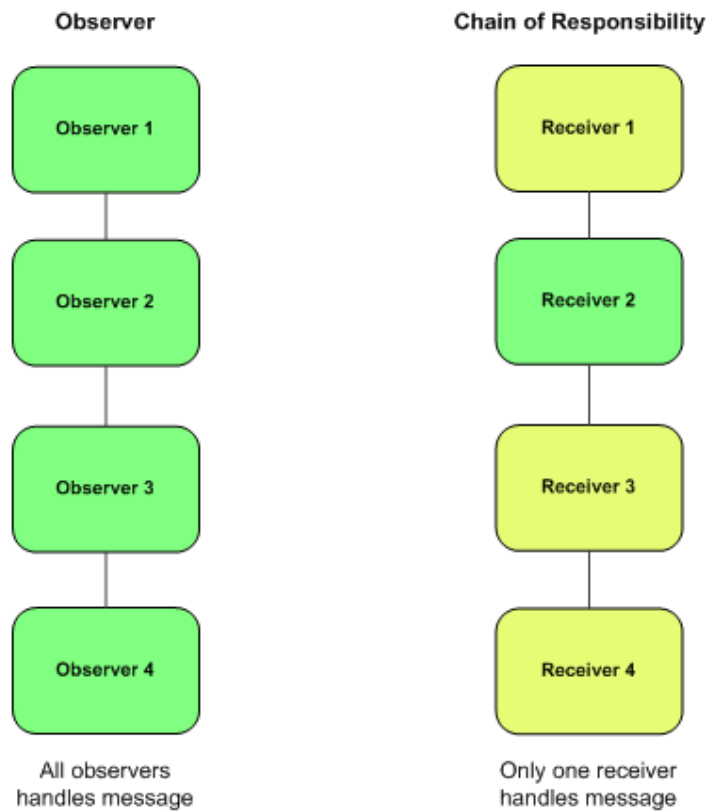
Following code depicts the above chain creation:

```
Logic logic = new Logic();
logic.Receiver = new Logger(new Emailer(null));
```

Note: Please note that the advantage of this pattern is decoupling of sender and receiver. The sender does not think about the right receiver. Instead it will pass the request and the appropriate receiver should process it. The above example can be written using a list of receivers in the Logic class with a Boolean property mentioning whether the message is processed or not.

Comparing Observer and Chain of Responsibility

In the case of Observer pattern all the registered receivers will get the request. Each of the receivers is interested in processing it. But in chain of responsibility the request is passed until it is not processed.



Summary

In this chapter we have explored Chain of Responsibility pattern with a C# example. The associated source code contains the example we discussed.

11. Abstract Factory Pattern

In this chapter I would like to explain the Abstract Factory pattern. This pattern is essential for learning the Factory Method and Bridge patterns.

Challenge

You are working on an ORM (Object Relational Mapping) framework creation. The framework could be used in different databases like:

- Sql Server
- Oracle
- MS Access
- OleDb

Based on the database type, you need to create related classes like:

- SqlConnection, SqlCommand when Sql server
- OracleConnection, OracleCommand when Oracle etc.

How to create a family of related objects without implementation of them?

Definition

GoF Definition: "Provide an interface for creating families of related or dependent objects without specifying their concrete classes"

Implementation

Our idea is to create an interface which contains all related objects like:

```
public abstract class DbProviderFactory
{
    public abstract DbConnection CreateConnection();
    public abstract DbCommand CreateCommand();
}
```

Luckily, ADO.NET already includes the same interface as Abstract Factory. The The definition is given below: *(I have excluded some irrelevant methods from it)*

```
namespace System.Data.Common
{
    public abstract class DbProviderFactory
    {
        public virtual DbCommand CreateCommand();
        public virtual DbCommandBuilder CreateCommandBuilder();
        public virtual DbConnection CreateConnection();
    }
}
```

```

        public virtual DbDataAdapter CreateDataAdapter();
        public virtual DbParameter CreateParameter();
    }
}

```

From the above code we can see that the related classes are included as a family without going to the implementation details.

The abstract class `DbProviderFactory` contains related classes which are abstract: `DbCommand`, `DbConnection` etc.

Concrete Implementations

ADO.NET provides the following implementations of the above abstract class `DbProviderFactory`.

- `System.Data.SqlClient.SqlClientFactory`
- `System.Data.OracleClient.OracleClientFactory`

The above classes implement all abstract methods from the base class. Thus it will be providing concrete implementations of `DbConnection` which is `SqlConnection`, `DbCommand` which is `SqlCommand` etc.

Instantiating from the above classes, we can start using the concrete implementations like following code.

```
SqlClientFactory factory = SqlClientFactory.Instance;
```

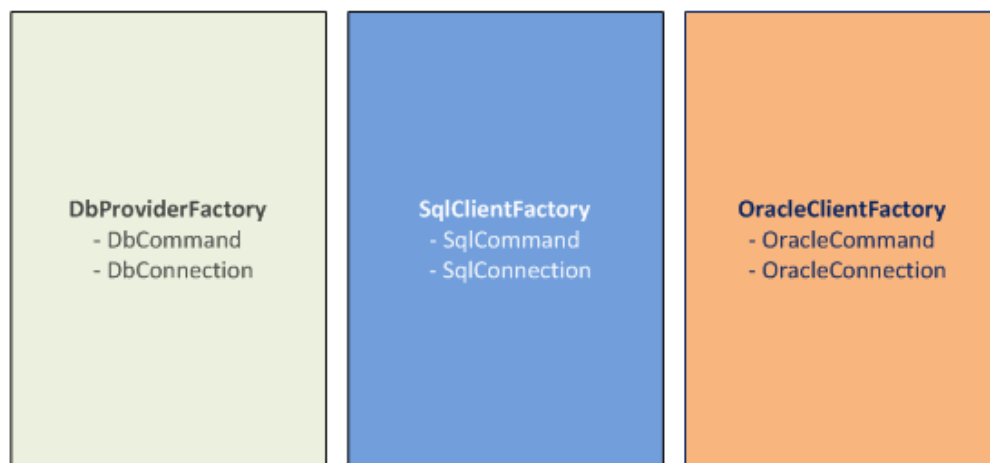
```
DbConnection connection = factory.CreateConnection();
```

```
DbCommand command = factory.CreateCommand();
```

```
command.Connection = connection;
```

```
command.CommandText = "query here";
```

```
command.ExecuteNonQuery();
```



Summary

In this chapter we have seen the usage of Abstract Factory pattern through inbuilt ADO.NET DbProviderFactory class. In the next chapter of Factory Method pattern the above discussed code will be used. So I am not attaching any source code with this chapter.

12. Factory Method Pattern

In the previous chapter you might have read about Abstract Factory. It says about abstracting a class with related objects. In this chapter we are discussing on Factory Method pattern. I am using the same ADO.NET Provider Factory class as example. Let us see the challenge and evolve to the solution.

Challenge

You are working on a windows application. The application has 2 types of users based on the database license they have: Sql Server or Oracle.

So for each database operation you need to create the right database classes based on an enumeration DatabaseType.

```
public void InsertRecord()
{
    if (AppContext.DatabaseType == DatabaseType.Sql Server)
    {
        SqlConnection connection = new SqlConnection();
        SqlCommand command = new SqlCommand();

        command.Connection = connection;
        command.ExecuteNonQuery();
    }
    else if (AppContext.DatabaseType == DatabaseType.Oracle)
    {
        OracleConnection connection = new OracleConnection();
        OracleCommand command = new OracleCommand();

        command.Connection = connection;
        command.ExecuteNonQuery();
    }
}
```

The above code seems to be complex and needs repeating for each database operation spread throughout the application.

How to stop the repeating code and make things better?

Definition

GoF Definition: "Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses"

Implementation

The above problem can be resolved using Factory Method pattern. Here we provide an abstract class for defining the structure and associating all related objects (using Abstract Factory). Then the sub classes will be created deriving from this abstract class. The sub classes decide which classes to be instantiated.

The advantage is more quality and less code. Here we have to define an abstract class named `DbProviderFactory`. (*already ADO.NET contains it*)

```
public abstract class DbProviderFactory
{
    public virtual DbCommand CreateCommand();
    public virtual DbCommandBuilder CreateCommandBuilder();
    public virtual DbConnection CreateConnection();
    public virtual DbDataAdapter CreateDataAdapter();
    public virtual DbParameter CreateParameter();
}
```

Then from the above abstract class we can derive concrete classes.

```
public class SqlConnectionFactory : DbProviderFactory
{
    public override DbConnection CreateConnection()
    {
        return new SqlConnection();
    }

    public override DbCommand CreateCommand()
    {
        return new SqlCommand();
    }
}
```

You can see from the above code `SqlConnection()` and `SqlCommand()` objects are created and supplied for `DbConnection` and `DbCommand` types respectively. Here `DbConnection` and `DbCommand` are other abstract classes.

Similarly the Oracle implementation follows:

```
public class OracleClientFactory : DbProviderFactory
{
    public override DbConnection CreateConnection()
    {
        return new OracleConnection();
    }

    public override DbCommand CreateCommand()
    {
        return new OracleCommand();
    }
}
```

Replacing the Original Code

Now we are ready to replace our `Insert()` method code with the Factory Method classes. Here we are using an `AppInit()` code to create the factory which will be used throughout the application.

```
public void AppInit()
{
    if (AppContext.DatabaseType == DatabaseType.SqlServer)
        AppContext.DbProviderFactory = SqlConnectionFactory.Instance;
}
```

```

        else if (AppContext.DatabaseType == DatabaseType.Oracle)
            AppContext.DbProviderFactory = OracleClientFactory.Instance;
    }

    // NewInsertRecord method
    private void NewInsertRecord()
    {
        DbConnection connection = AppContext.DbProviderFactory.CreateConnection();
        DbCommand command = AppContext.DbProviderFactory.CreateCommand();

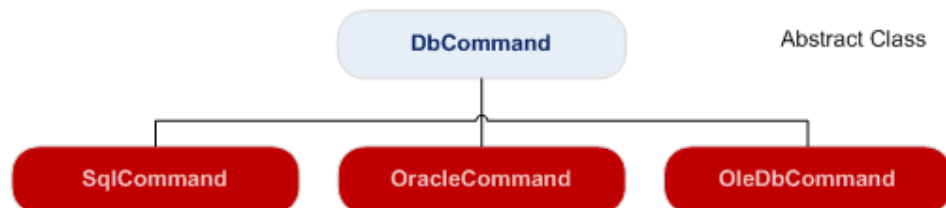
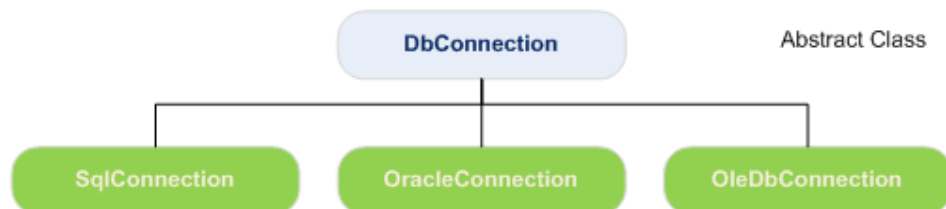
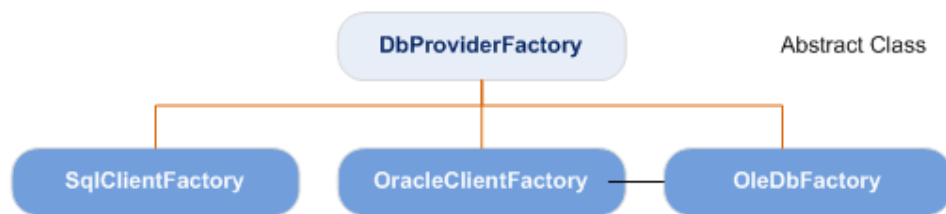
        command.Connection = connection;
        command.ExecuteNonQuery();
    }

```

From the above code you can see that using Factory Method the code is reduced considerably but at a cost of new classes. The pattern provides much flexibility on adding a new database. (if another customer with PostgreSQL arrives 😊)

Abstract and Concrete Derivation

The following image depicts the relation between the abstract and concrete classes we have discussed.



Note

ADO.NET already includes the **DbProviderFactory** abstract class and concrete classes like **SqlClientFactory**, **OleDbFactory** etc. Additionally, using an ORM like Entity Framework automatically takes care of the database switching. The scenario mentioned here is for learning purposes.

Summary

In this method we have seen the Factory Method pattern. The example we discussed is included with the attachment.

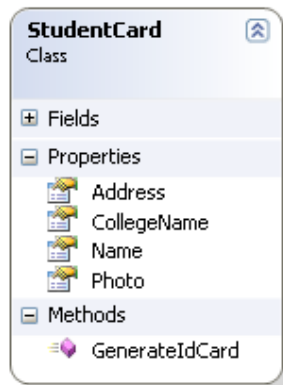
13. Flyweight Pattern

In this chapter we are discussing about Flyweight Design Pattern. It is one among the 23 design patterns and provides an improved way of managing objects.

Challenge

You are working on an Id Card Creation web application. The application has 1 lakh Student records for which Id Cards to be created.

Following is the StudentCard class that performs the id card generation. We need to call the GenerateIdCard() method to get the Id Card image after assigning the student Name, Address, Photo properties. Each card image generated is saved to file system.



The problem is the 1 Lakh class instances created. How to reduce the number of instances?

Definition

GoF Definition: "Use sharing to support large numbers of fine-grained objects efficiently"

Implementation

Using Flyweight pattern we can solve the above problem. We can see from the above problem that at a time only one instance is needed.

We can use only one instance of StudentCard class and share it inside the loop to assign the properties and generate id card.

The pattern is advisable on:

- To reduce the number of instances
- Sharing properties which are common

This was the old process:



```
For Loop  
    Create Instance of StudentCard  
    Assign Properties  
    Invoke GenerateIdCard()  
End Loop
```

Here depending on the Student count, instances of StudentCard are created.

This will be the new process:



```
    Create Instance of StudentCard  
For Loop  
    Assign Properties  
    Invoke GenerateIdCard()  
End Loop
```

Here only 1 instance of StudentCard is created.

Code View

Following is the code implementing Flyweight pattern:

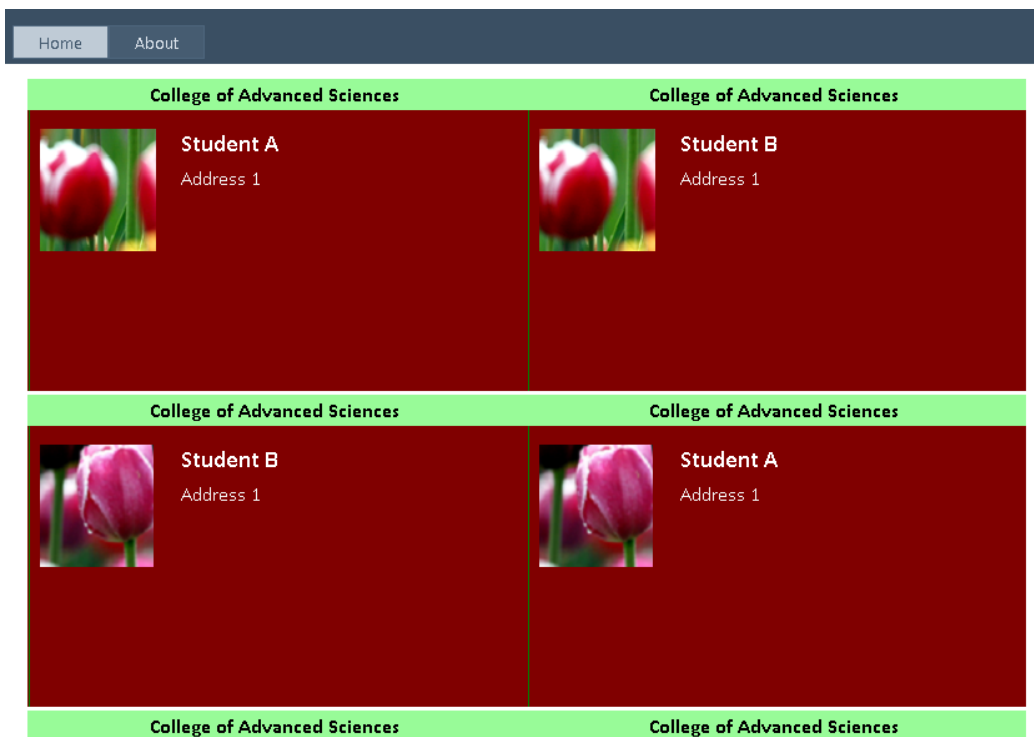
```
StudentCard card = new StudentCard();
card.CollegeName = "College of Advanced Sciences"; // CommonProp

for (int i = 1; i <= 100; i++)
{
    card.Name = names[random.Next(0, names.Length - 1)];
    card.Address = addresses[random.Next(0, addresses.Length - 1)];
    card.Photo = photos[random.Next(0, photos.Length - 1)];

    list.Add(new System.Web.UI.WebControls.Image()
    {
        ImageUrl = card.GeneratedCard()
    });
}
```

Screen Shot

On running the attached web application you can see the following output. The data like Name, Address and Photo are randomly generated.



Drawbacks

Although the Flyweight pattern solves many problems, I would like to list some of the possible drawbacks of using it.

- More configuration code is needed to switch between properties
- Reduces performance in a multi-threaded environment if locks are used

Summary

In this chapter we have explored Flyweight design pattern. The pattern helps us in reducing system resources if correctly used. The associated source code contains the example we have discussed. Please let me know your comments on the chapter.

14. Proxy Pattern

In this chapter we are going to discuss about Proxy design pattern. It is one among the 23 design patterns by Gof. As usual we can start with the Challenge and Solution style.

Challenge

You are working on automating Excel COM objects. The business logic has to think too much about instantiating COM, doing ground works before calling the actual functionalities etc.

The same code could get replaced by DCOM tomorrow. So the Excel access code is spread throughout your application. How to do a better design?



Definition

GoF Definition: "Provide a surrogate or placeholder for another object to control access to it"

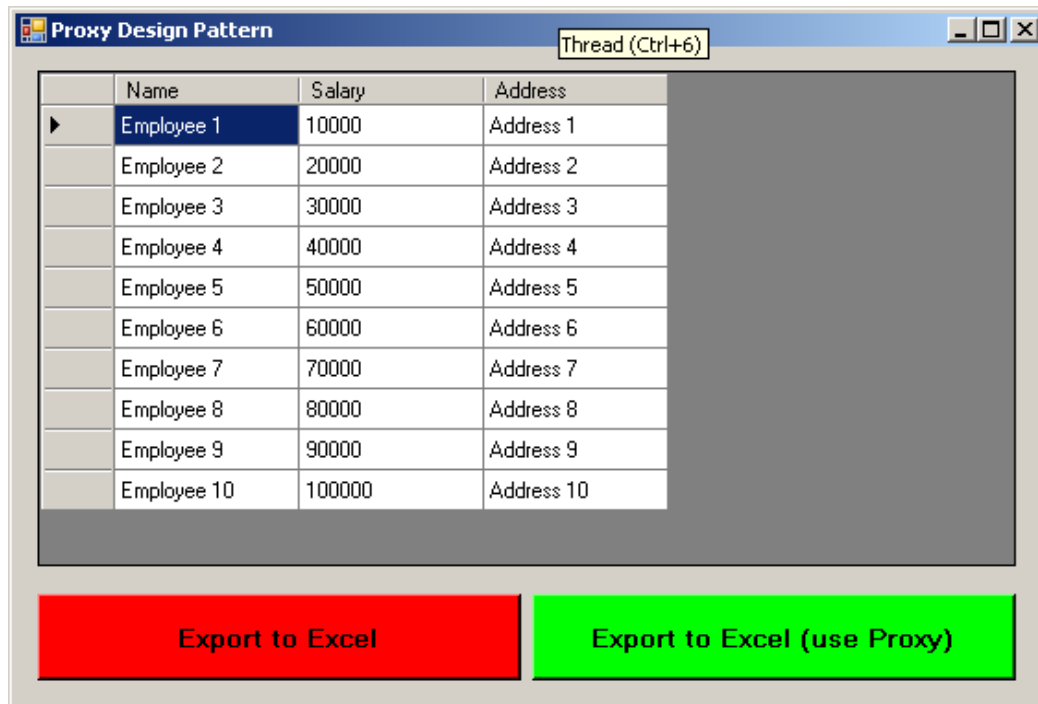
Implementation

We can solve the above problem by using a Proxy pattern. As the definition says, we will have to create a placeholder or wrapper around the original object to control access to it. In this way we can make the following advantages:

- Talk to COM or DCOM Excel object by changing configuration in one place
- Give application a simple interface to talk with Excel COM/DCOM
- Feel application think like it is talking to a local object

Application

Following is the application with data:



Old Code

Following is the old code where application is forced to think too much about the Excel COM object and method of assigning values to the cells.

```
//Start Excel and get Application object.
Microsoft.Office.Interop.Excel.Application oXL = new
Microsoft.Office.Interop.Excel.Application();
oXL.Visible = true;

//Get a new workbook.
Microsoft.Office.Interop.Excel.Workbook oWB =
(Microsoft.Office.Interop.Excel.Workbook)(oXL.Workbooks.Add(Missing.Value));
Microsoft.Office.Interop.Excel.Worksheet oSheet =
(Microsoft.Office.Interop.Excel.Worksheet)oWB.ActiveSheet;

//Add table headers going cell by cell.
oSheet.Cells[1, 1] = "Name";
oSheet.Cells[1, 2] = "Address";
oSheet.Cells[1, 3] = "Salary";

//Format A1:D1 as bold, vertical alignment = center.
oSheet.get_Range("A1", "C1").Font.Bold = true;
oSheet.get_Range("A1", "C1").VerticalAlignment =
Microsoft.Office.Interop.Excel.XlVAlign.xlVAlignCenter;

int i = 2;
foreach (Employee employee in _list)
{
    string[] values = new string[3];
    values[0] = employee.Name;
    values[1] = employee.Address;
    values[2] = employee.Salary.ToString();
}
```

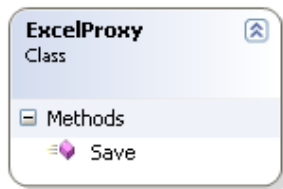
```
oSheet.get_Range("A" + i.ToString(), "C" + i.ToString()).Value = values;  
i++;  
}  
  
oXL.Visible = true;  
oXL.UserControl = true;
```

The problem with above code is too much Excel logic is mixed with the application logic. Now we can see the new code with proxy pattern implemented.

New Code with Proxy Pattern

```
ExcelProxy proxy = new ExcelProxy();  
proxy.Save(list);
```

The code is only 2 lines and the Excel COM object creation, cell value assigning etc are taken care by the ExcelProxy class.



Application Execution

On executing the application we can see the results inside Microsoft Excel.

Book3 - Microsoft Excel

Home Insert Page Layout Formulas Data Review View Load Telemetry

Clipboard Font Alignment Number Styles Cells Editing

G4

	A	B	C	D
1	Name	Address	Salary	
2	Employee 1	Address 1	10000	
3	Employee 2	Address 2	20000	
4	Employee 3	Address 3	30000	
5	Employee 4	Address 4	40000	
6	Employee 5	Address 5	50000	
7	Employee 6	Address 6	60000	
8	Employee 7	Address 7	70000	
9	Employee 8	Address 8	80000	
10	Employee 9	Address 9	90000	
11	Employee 10	Address 10	100000	
12				

Sheet1 Sheet2 Sheet3

Ready 100%

Other Examples of Proxy Pattern

We can have many real world examples which implement the Proxy pattern. When we add a WCF reference a Proxy is created. This class takes care of the connection details, serialization etc.

Summary

In this chapter we have seen the usage of Proxy design pattern along with an example. The source code attached contains the example we have discussed.

15. Facade Pattern

In this chapter I would like to explore the Façade design pattern. The actual pronunciation is fu'saad. The dictionary meaning of the word Façade is "A showy misrepresentation intended to conceal something unpleasant"

Now let us enter the challenge..

Challenge

You are working on a database application. Frequently you need to execute UPDATE and DELETE SQL queries. Each time there is a need to create the SqlConnection and SqlCommand class, assign the connection to command and execute the query.

The series of activities looks like:

```
SqlConnection connection = new SqlConnection("connection string");
SqlCommand command = new SqlCommand();
command.Connection = connection;
command.CommandText = "UPDATE Customer SET Processed=1";
command.ExecuteNonQuery();
```

You need to repeat the same code wherever execute queries are required. How to make this code better?

Definition

GoF Definition: "Provide a unified interface to a set of interfaces in a system. Facade defines a higher-level interface that makes the subsystem easier to use"

Implementation

Using Façade, we can improve the situation. We can find only the query is different in each case – the parameters like connection string is common for the entire application.

```
SQLFacade.ExecuteNonQuery("UPDATE query here..");
```

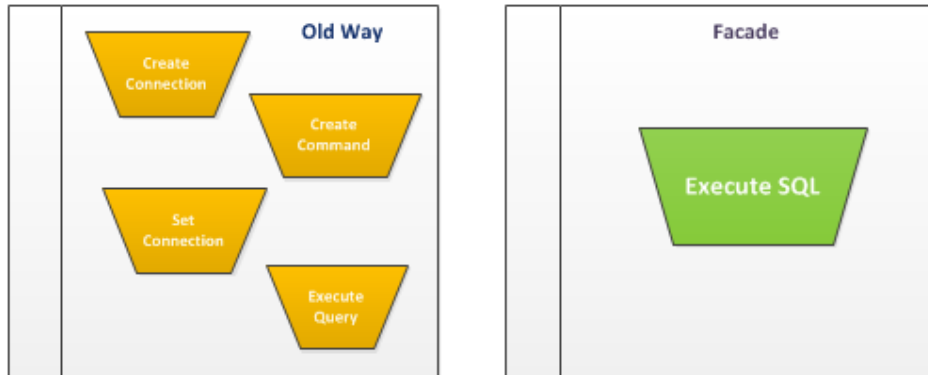
After using the Façade the code will look like above.

The complicated code is being pulled to the background SQLFacade class.

```
namespace FacadePattern
{
    public class SQLFacade
    {
        public static bool ExecuteSQL(string sql)
        {
            SqlConnection connection = new SqlConnection("connection string");
            SqlCommand command = new SqlCommand();
            command.Connection = connection;
            command.CommandText = sql;
            command.ExecuteNonQuery();
        }
    }
}
```

```
        return true;  
    }  
}
```

Façade pattern makes code better as depicted in the image below:



Summary

In this chapter we have seen how to use the Façade pattern to improve our code. It is similar to the use of reusable functions and it pulls out the complications and provides an easier interface for use. The associated code contains the classes we have discussed.

16. State Pattern

In this chapter I am going to explain the State Pattern. It is one among the 23 design patterns and provides a good solution to a common problem. As usual the pattern starts with a challenge and then implementing the solution using the pattern.

Challenge

You are working on a job processing application. The application can handle only 1 job at a time. So when a new job arrives the system response will be weird. How to provide a better approach to the above problem?

Definition

Gof Definition: "Allow an object to alter its behavior when its internal state changes. The object will appear to change its class"

Implementation

An enumeration named JobState is used to define the states used in our application.

```
public enum JobState
{
    Ready,
    Busy
}
```

This enumeration is used inside our Job class.

```
public class Job
{
    private JobState _State;

    public JobState State
    {
        get { return _State; }
        set
        {
            _State = value;

            if (OnJobStateChanged != null)
                OnJobStateChanged(this, _State);
        }
    }

    public bool AddJob(object jobParameter)
    {
        switch (State)
        {
            case JobState.Ready:
            {
                DoJob();
                return true;
            }
            case JobState.Busy:
            {
```

```
        return false;  
    }  
    }  
    return false;  
}
```

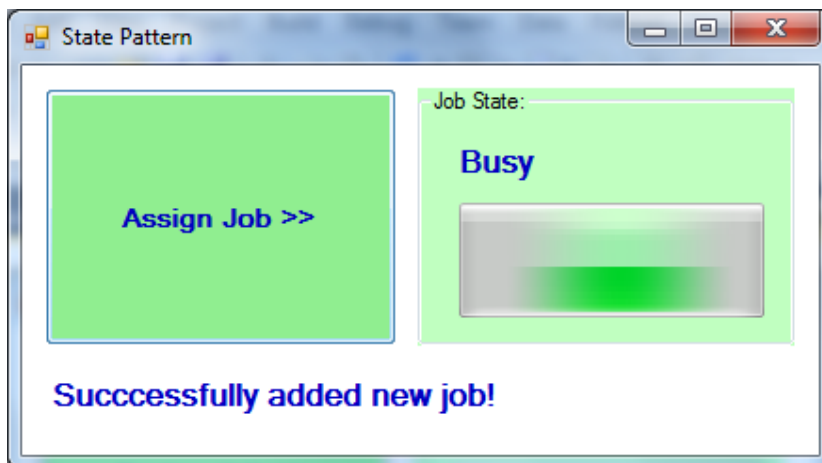
In the AddJob() method we can see the new job is taken only if the JobState is in Ready state. Here true will be returned by the method. If there is another job running, then the new job will not be queued and false will be returned.

Thus the object Job changes the behavior based on the internal state.

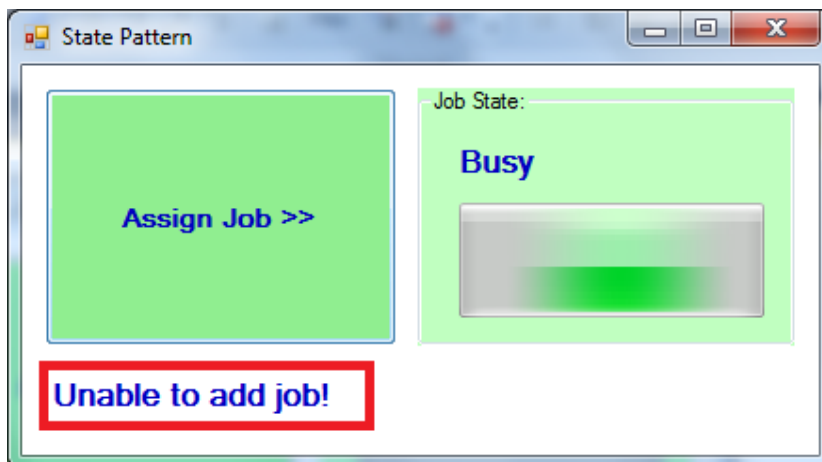
Application

The associated application contains the classes we discussed.

On executing the windows application and clicking the Assign Job button, you can see the following screen:



While a job is in process, we cannot add another job. If a try is made the response is:

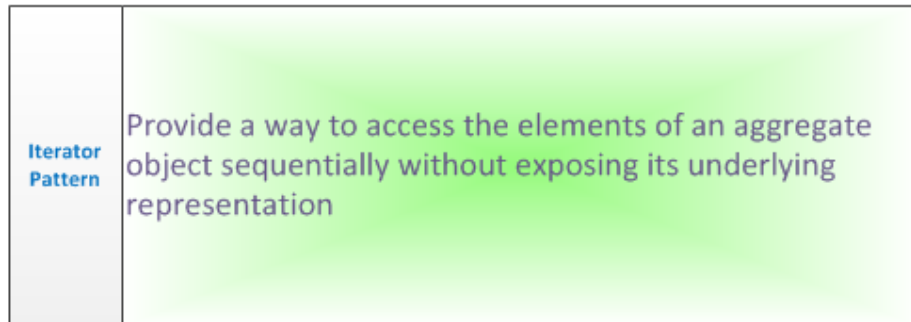


Summary

In this chapter we have seen about State Pattern. The attachment file contains the source code of the application we have discussed.

17. Iterator Pattern

In this chapter we are going to discuss about the Iterator design pattern. Most of the Object Oriented Languages nowadays supports this pattern through their core language infrastructure.



Challenge

You are working on a Name printing application. You have to deal with a Bank class with the following properties.

```
public class Bank
{
    public string ManagerName
    {
        get;
        set;
    }

    public string AccountantName
    {
        get;
        set;
    }

    public string CashierName
    {
        get;
        set;
    }
}
```

For printing the name the Print() method is called as following:

```
Print(_bank.Manager);
Print(_bank.Accountant);
Print(_bank.Cashier);
```

Now there is an addition of Branch type Bank to the above Bank object. Now the situation has become more complex. How to do a better approach?

Definition

GoF Definition: "Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation"

Implementation

We can improve the above Challenge Situation using the Iterator pattern. Instead of calling Print() method each time, we can use an Iterator to get the names inside Bank Object.

We are introducing a new class named **BankIterator** which implements IEnumerable:

```
public class BankIterator : IEnumerable
{
    private IList _list = new ArrayList();

    public BankIterator(Bank bank)
    {
        _list.Add(bank.Manager);
        _list.Add(bank.Accountant);
        _list.Add(bank.Cashier);
    }

    public IEnumerator GetEnumerator()
    {
        return _list.GetEnumerator();
    }
}
```

The class uses an ArrayList to hold the Manager, Accountant, Cashier names. As it is inheriting IEnumerable it has to implement the GetEnumerator() method. Luckily we can send the list.GetEnumerator() for the same.

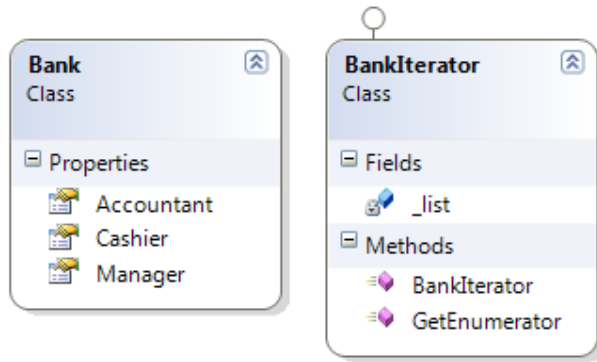
The new Print() method invocation looks like below:

```
BankIterator iterator = new BankIterator(_bank);
foreach (string name in iterator)
    Print(name);
```

This improves the situation in the following ways:

- Easier Names access through Iteration
- More flexibility in introducing new Name properties in the Bank class

Following is the snapshot of Class Diagrams:



Note

From C# 2.0 onwards the Iterator pattern is implemented using **foreach**. The **foreach** allows sequential accessing of the elements inside a collection. The **IEnumerator** interface is used to implement the Iterator pattern.

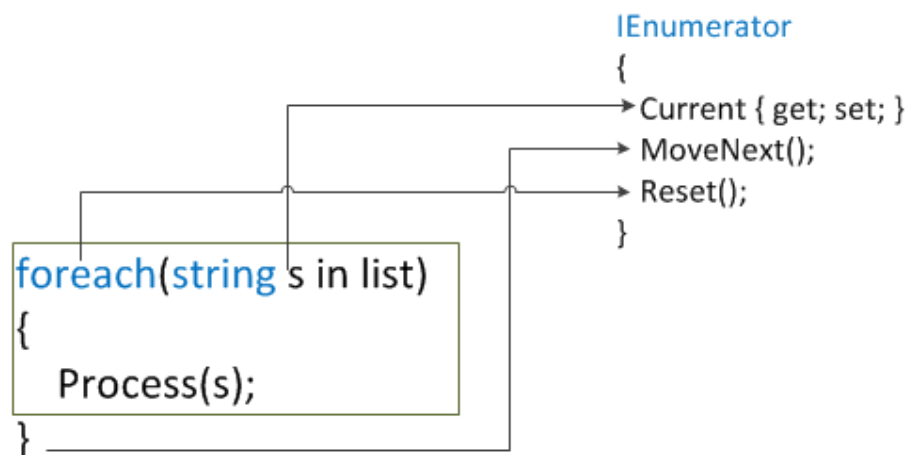
```

public interface IEnumerator
{
    object Current { get; }

    bool MoveNext();

    void Reset();
}
  
```

While executing the **foreach**, first the **Reset()** method is called to position to the first element of the collection. Then through each iteration the **MoveNext()** method is called. The **Current** property returns the current object from the collection. The **IEnumerator** interface implementation of array, List, HashSet etc. enables us to iterate over them using the same **foreach** loop.



References

[http://msdn.microsoft.com/en-us/library/dscyy5s0\(v=vs.80\).aspx](http://msdn.microsoft.com/en-us/library/dscyy5s0(v=vs.80).aspx)

Summary

In this chapter we have explored the Iterator pattern with an example in C#. The pattern is already well implemented inside the language architecture of C#. We can extend the pattern usage in our custom classes too. The examples we discussed above are included in the source code attachment.

18. Mediator Pattern

In this chapter I would like to take you through the advantage of using Mediator Pattern. As always, Design Patterns if properly used gives us more flexibility and manageability. Mediator Pattern is of no deviation from this property.

Challenge

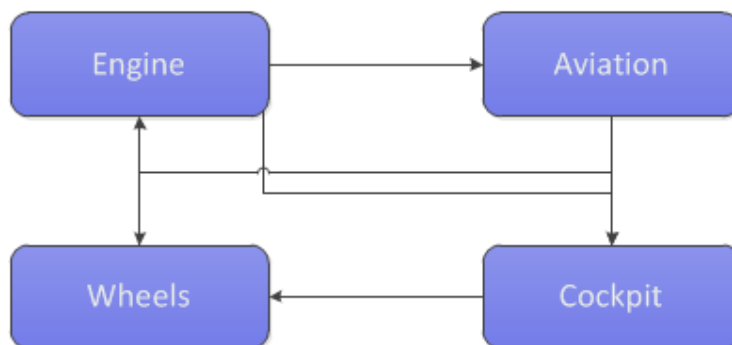
You are working on an Airline application managing flight validations. As you might know before taking off the flight, it goes through a series of checks ensuring the health of each component, the fuel level etc.

Our Flight class contains the following component classes

- Engine
- Wheels
- Cockpit
- Aviation

Each of the component class will be having a method named `Start()`. On invoking the method, it will ensure that the current state of component is valid through a method named `IsReady()`. If the `IsReady()` method returned true, the component class checks the other component `IsReady()` method.

For example the Engine class ensures the Wheels and Aviation is in valid statue by using the `IsReady()` method of Wheel and Aviation respectively. The Cockpit in turn checks itself, Engine and the Aviation status. So there exists a coupling between the classes and code duplications. The scenario will become worse on introducing new components as classes.



How to make the code better?

Definition

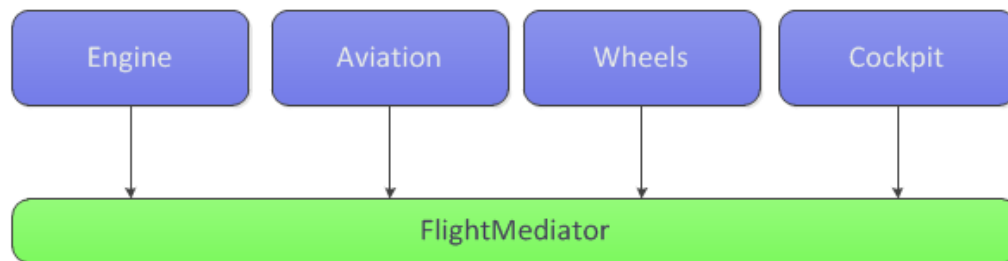
GoF Definition: "Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently"

Implementation

We can improve the above situation by introducing the Mediator pattern. Through the introduction of Mediator the component classes won't talk to each other. All the components communicate to a new Mediator class. The advantages are:

- Loosely coupled components
- Centralized Management
- More Flexibility in changing code

The new approach will look like below:



Here the component parts will communicate with the Mediator class for providing the ready signal to proceed with.

Classes

Following are the classes associated with the new implementation using Mediator Pattern.

```

public class FlightMediator
{
    private Engine _engine;
    private Aviation _aviation;
    private Wheels _wheels;
    private Cockpit _cockpit;

    public FlightMediator(Engine engine, Aviation aviation, Wheels wheels,
        Cockpit cockpit)
    {
        _engine = engine;
        _aviation = aviation;
        _wheels = wheels;
        _cockpit = cockpit;
    }
}
  
```

```

    }

    public bool IsReady()
    {
        return _engine.IsReady() && _aviation.IsReady() && _wheels.IsReady() &&
        _cockpit.IsReady();
    }
}

public class Engine
{
    public void Start()
    {
    }

    public bool IsReady()
    {
        return true;
    }
}

public class Aviation
{
    private int _FuelLevel = 1000;

    public bool IsReady()
    {
        return _FuelLevel > 5000; // Returns false as not enough fuel
    }
}

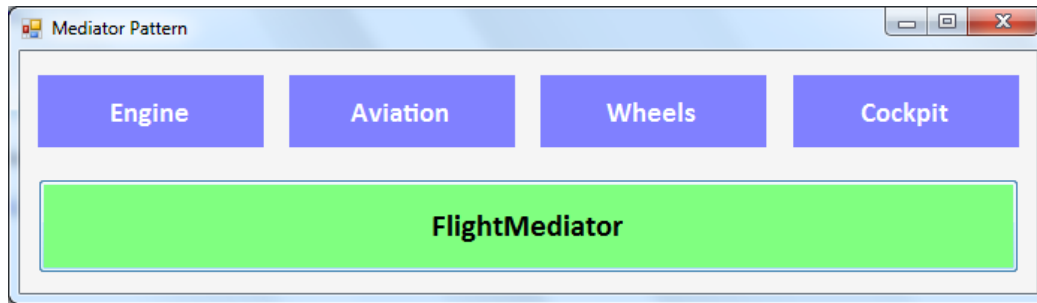
public class Wheels
{
    public bool IsReady()
    {
        return true;
    }
}

public class Cockpit
{
    public bool IsReady()
    {
        return true;
    }
}

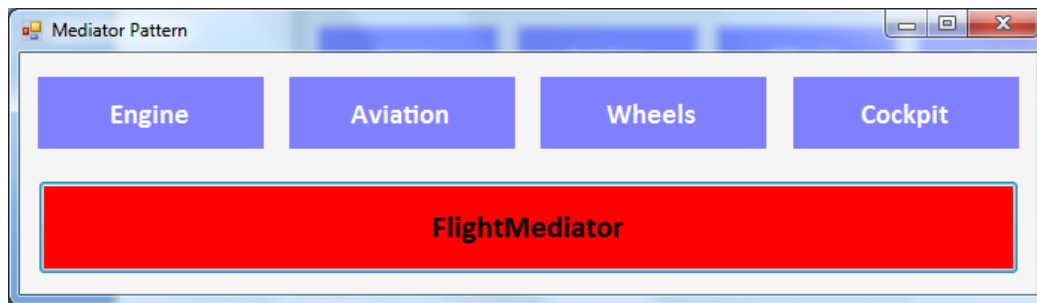
```

Please note that the Aviation class returns false for the IsReady() method. The FlightMediator constructor takes all the component classes as inputs. The IsReady() method checks all the associated component's IsReady() method to ensure validations are right.

On running the Windows Forms application, you can see the following screen.



Clicking on the FlightMediator button you can see the color changing to red as the status returned is false.



Summary

In this chapter we have explored the Mediator pattern. The example is provided in a simple problem scenario, but the real world problems will be much complicated. For example a Wizard Page Framework where each page has to think about the previous and next pages could be made better using the Mediator pattern. The associated source code is attached with the chapter.

19. Memento Pattern

In this chapter I would like to experiment with the Memento pattern. Memento pattern provides an Object Oriented way of saving the state of an object. The state called as Memento can be used to restore the object later.

The English meaning of Memento is Reminder of Past Events.

Challenge

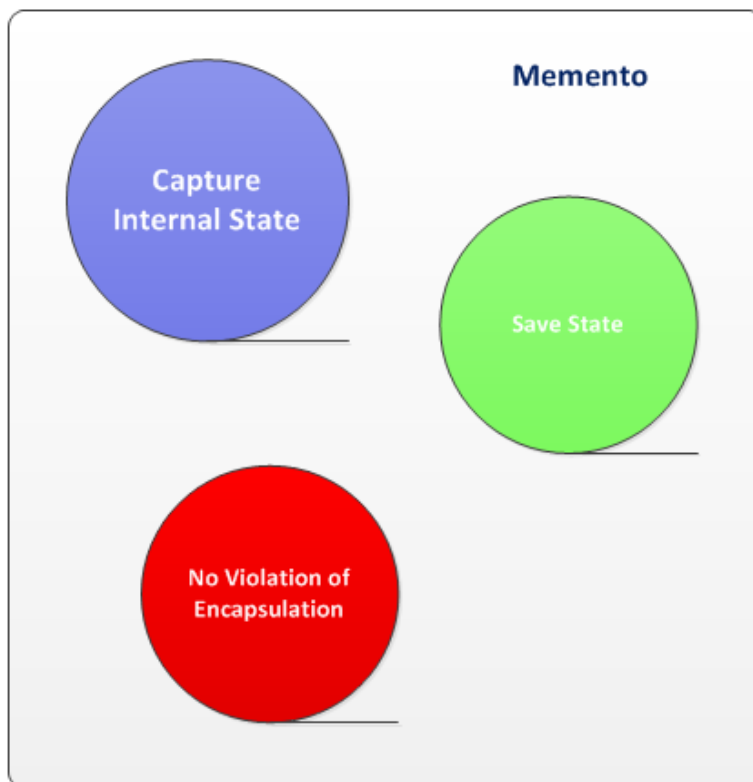
You are working on a Drawing application which allows the user to draw lines on a Canvas. The user should be able to Save the state on particular intervals so any further mistakes could be undone. You need to provide a solution for the problem.

Definition

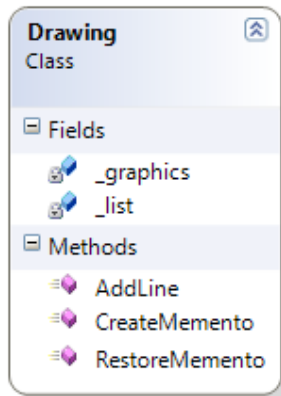
GoF Definition: "Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later"

Implementation

The above problem can be solved using the Memento Pattern. This pattern as said in the definition saves the internal state of the object for restoring later. The object state is saved without violating Encapsulation.



Following is the Drawing class definition:

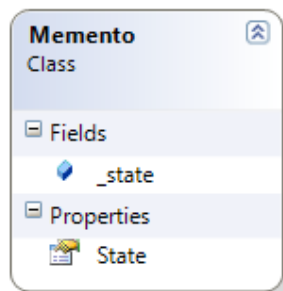


The `AddLine()` method allows to add line on the graphics. These information are stored inside the `_list` object as state.

The `CreateMemento()` allows to get a copy of the internal state without violating Encapsulation.

The `RestoreMemento()` allows to restore the state to the given memento.

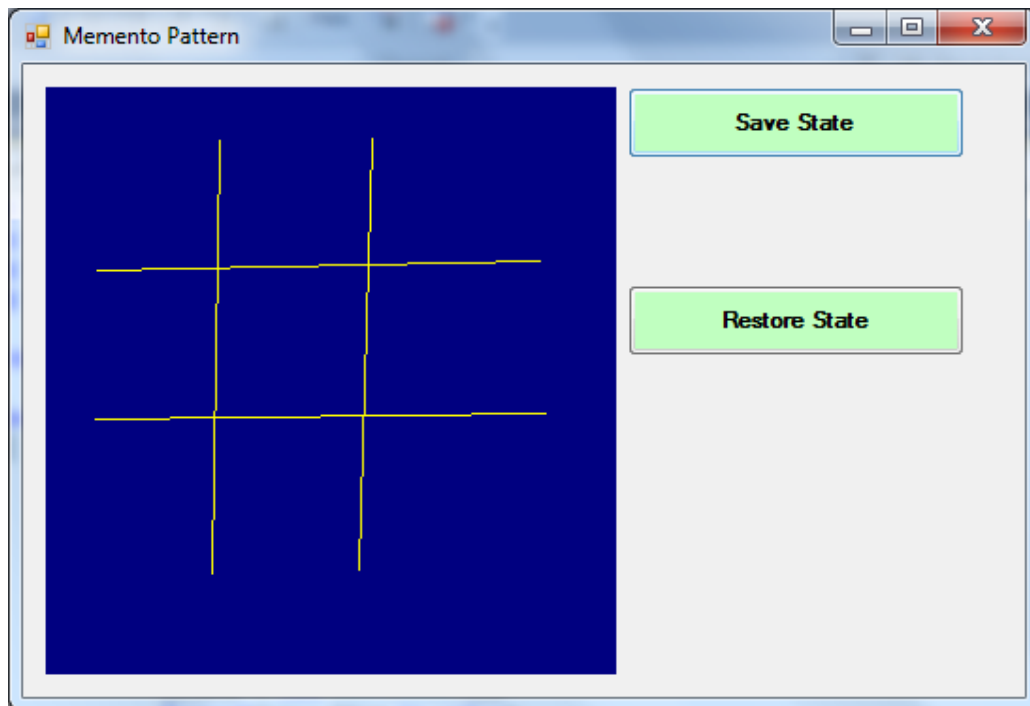
Following is the Memento class definition:



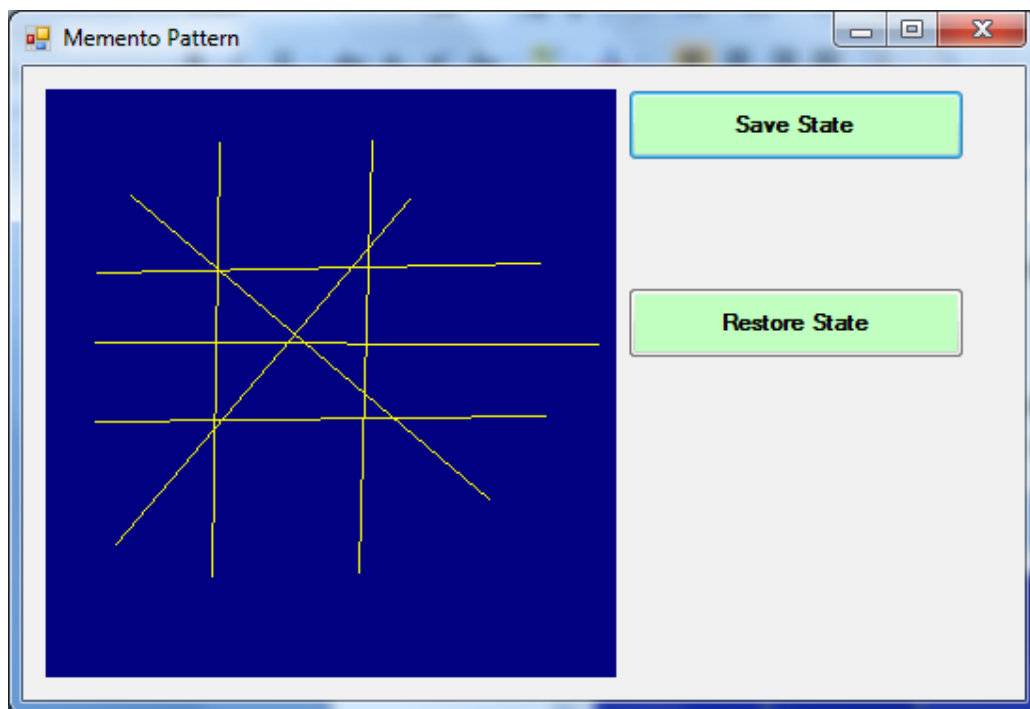
It contains the `State` property to hold the State of Drawing class.

Executing the Application

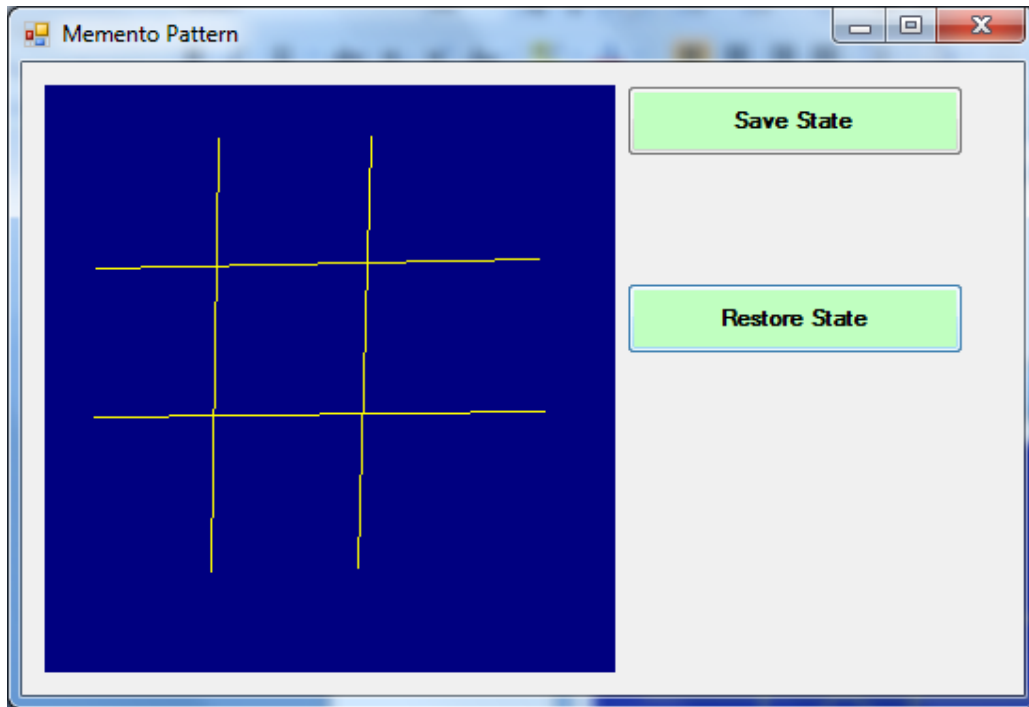
You can try executing the attached application. After executing draw a series of lines in the Canvas and use the Save button to save the memento.



Now try to add some unwanted lines:



Use the Restore State to get the old valid state.



So this concludes our application implementing Memento pattern.

Comparing Command and Memento Patterns

The Command pattern also provides the Undo / Redo functionality by converting actions to commands. The difference would be that the Command pattern stores each action but the Memento pattern saves the state only on request. Additionally, the Command pattern has Undo and Redo operations for each action, but the Memento does not need that.

Depending on the scenario, Command / Memento pattern could be used.

Summary

In this chapter we have explore the Memento pattern. The attached source code contains the example we discussed.

20. Prototype Pattern

In this chapter I would like to explain the Prototype Pattern and C# support for it. The prototype pattern provides an alternative to instantiating new objects by copying the prototype of an existing one.

Challenge

You are working on an application having setting stored in class Settings. The properties of the class are assigned inside the constructor. It involves calling the configuration file, security database, user profile database etc.

You need to pass the above instance to a component inside a library. The component could play around with the properties so we need to send only another instance of Settings. As the instance creation is expensive we need to create a copy of the above instance.

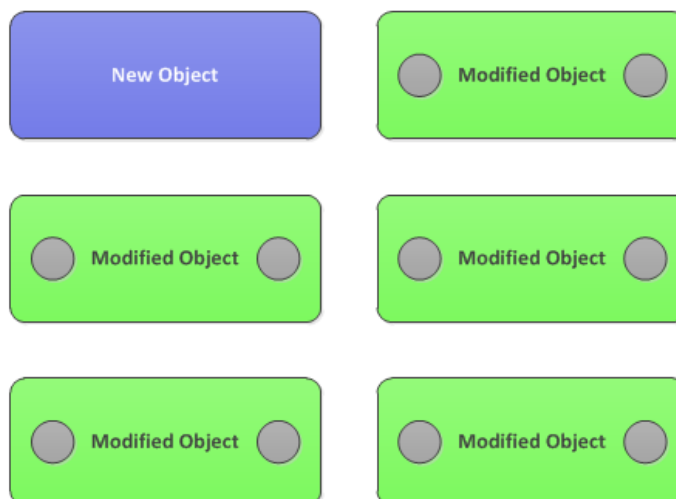
What would be the best approach?

Definition

GoF Definition: "Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype"

Solution

We can use the Prototype pattern to address this scenario. This pattern allows us to create a new instance by prototyping an existing instance with required properties.



Create new object and modify it. Create prototypes from modified object.

The System.Object class includes new method named `MemberwiseClone()` which provides a shallow copy of the current object. Shallow copy involves copying only

the value types. Coupled with ICloneable interface we can provide a Clone() method that invokes MemberwiseClone() for the Settings class.

```
public class Settings : ICloneable
{
    public Settings()
    {
        // Load ApplicationSettings from Configuration
        // Load ThemeSettings from Database
        // Load UserSettings from Database
        Thread.Sleep(3000); // Simulate a delay for the above operations

        ApplicationSettings = "ApplicationSettings1";
        ThemeSettings = "ThemeSettings1";
        UserSettings = "UserSettings1";
    }

    public string ApplicationSettings
    {
        get;
        set;
    }

    public string ThemeSettings
    {
        get;
        set;
    }

    public string UserSettings
    {
        get;
        set;
    }

    public object Clone()
    {
        return this.MemberwiseClone();
    }

    public override string ToString()
    {
        return this.ApplicationSettings + " " +
            this.ThemeSettings + " " +
            this.UserSettings;
    }
}
```

Executing the Application

Now we can try playing with the class instances. First we create an instance of Settings (which is a 3 second delay process). Later we take a clone of the Settings instance and modify the properties. We can see that the new instance was created using Clone() method.

```
static void Main(string[] args)
{
    Settings settings = new Settings();
    Settings settingsClone = (Settings)settings.Clone();
}
```

```
settingsCl one. ApplicationSettings = "NewSettings"; // Change property
```

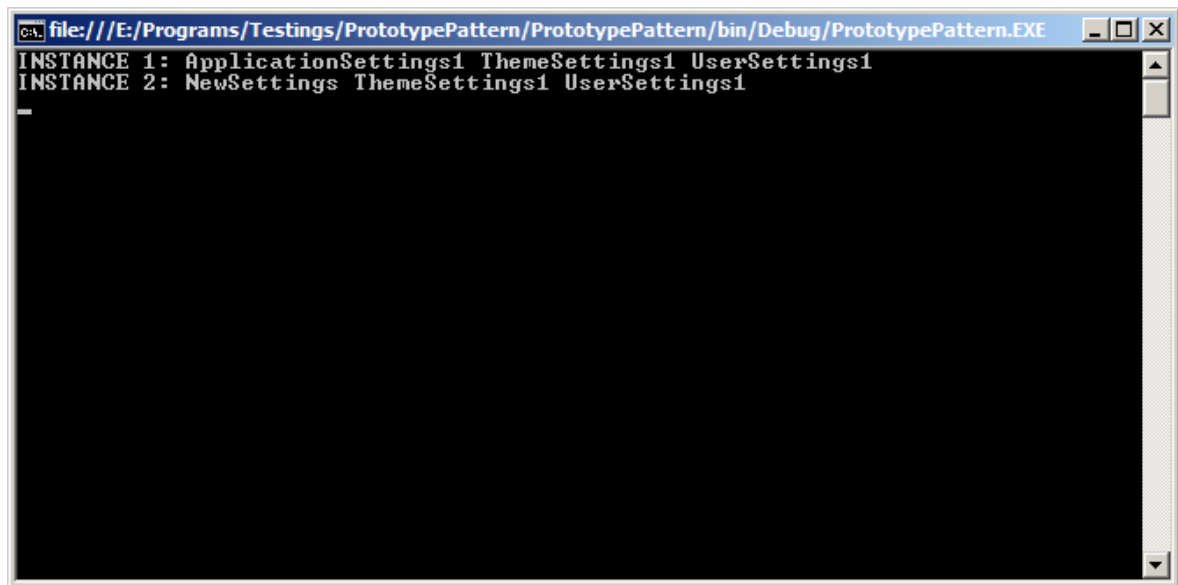
```
Console.WriteLine("INSTANCE 1: " + settings.ToString());
```

```
Console.WriteLine("INSTANCE 2: " + settingsCl one.ToString());
```

```
Console.ReadKey(false);
```

```
}
```

On executing the application we can see the following results.



We can see from the above code that:

1. Instance 1 was created using new keyword
2. Instance 2 was created using Clone() method
3. Changing of Instance 2 does not affect Instance 1

So this concludes our experiment with the Prototype pattern.

References

http://sourcemaking.com/design_patterns/prototype

Summary

In this chapter we have explored the Prototype pattern. The usage of Prototype pattern can be summarized as:

- Faster instance creation with required properties
- The new keyword can be avoided in instance creation
- Provides an alternative to Abstract Factory pattern
- Runtime specification of instance properties

- Reference of concrete class can be avoided

The associated source code contains the example we discussed.

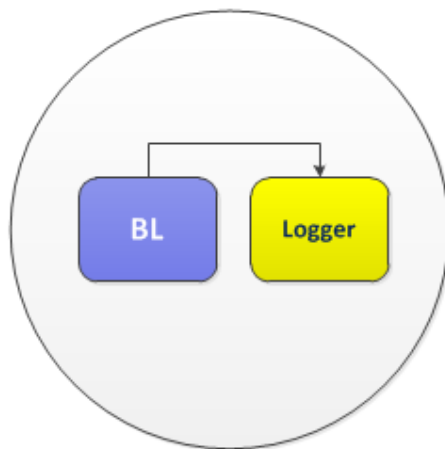
21. Bridge Pattern

In this chapter we can explore the Bridge pattern which is commonly used in applications.

Challenge

You are working on an abstract business class. The class needs to log information to by writing to a file, email to user etc. We need to avoid tightly coupling of the business class and logger as in future newer versions of logger might appear.

Tightly Coupled Design



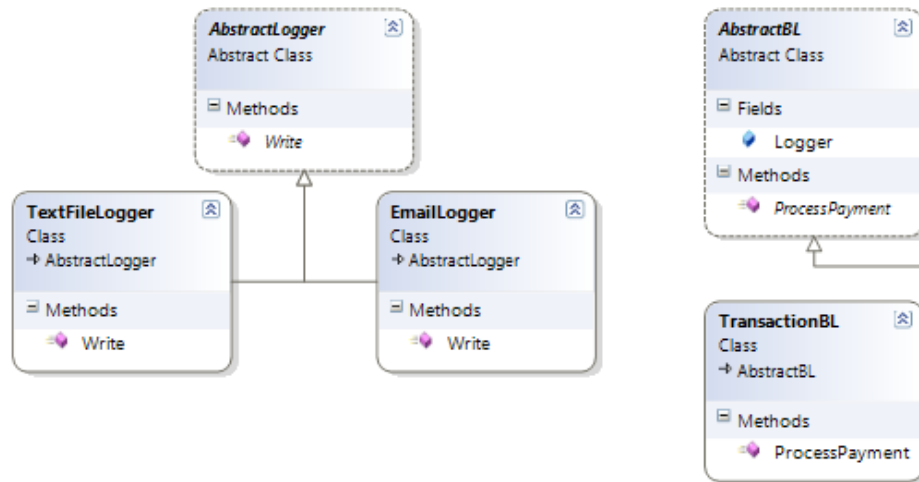
How to achieve a decoupled solution?

Definition

GoF Definition: "Decouple an abstraction from its implementation so that the two can vary independently"

Implementation

We can create an abstract class for representing the Logger. Let the abstract business class hold reference to the new abstract Logger. In the runtime we can specify the actual implementation instance of Logger class. (File writer or Emailer) Following is the snapshot of the class diagram after using the Bridge Pattern:



Here the concrete Logger implementations derive from AbstractLogger class. The concrete TransactionBL derives from AbstractBL class. The AbstractBL class is holding reference to Logger (abstract Logger).

Using the Bridge Pattern as shown above, there is decoupling of the implementation from its abstraction. It allows future extensibility to the Logger classes without modifying the BL class.

Code

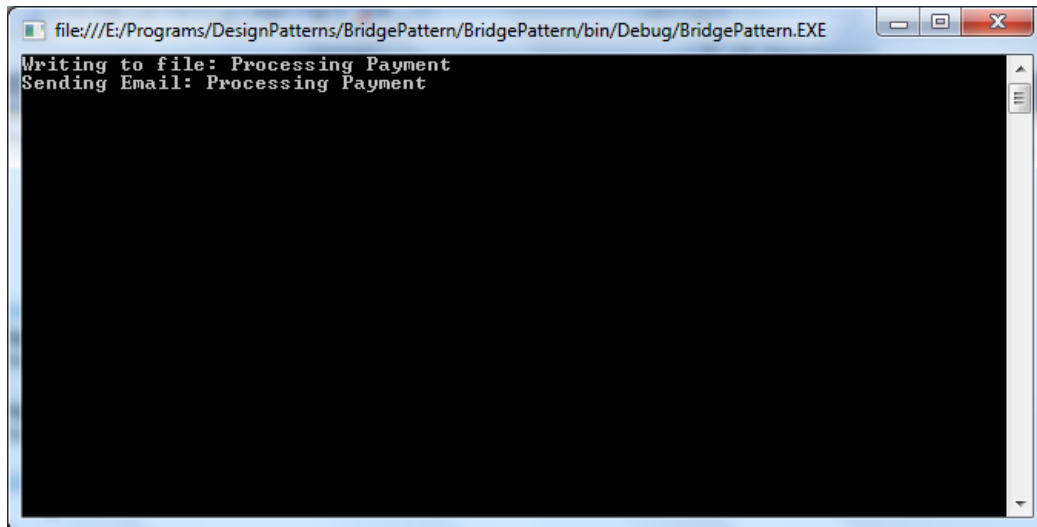
Following is the code that invokes above Transaction and Logger classes.

```
AbstractBL bl = new TransactionBL();
bl.Logger = new TextFileLogger();
bl.ProcessPayment();
```

```
bl.Logger = new EmailLogger();
bl.ProcessPayment();
```

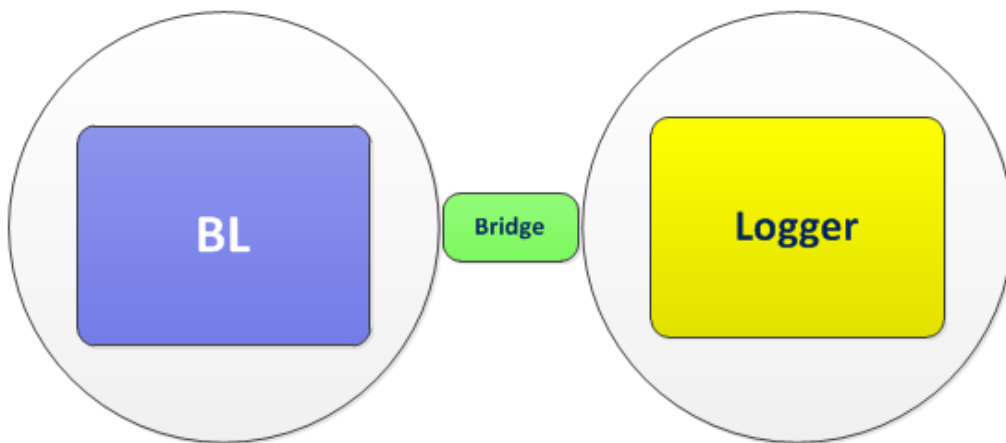
```
Console.ReadKey(false);
```

On running the application we can see the following output.



```
file:///E:/Programs/DesignPatterns/BridgePattern/BridgePattern/bin/Debug/BridgePattern.EXE
Writing to file: Processing Payment
Sending Email: Processing Payment
```

Following is the coupling snapshot after using the Bridge Pattern.



Note

We can see that this pattern is a simple implementation using an abstract class and interface. The Bridge Pattern is having some similarity with Strategy Pattern as both involve changing the implementation in the runtime. Here the class is abstract when compared with Strategy pattern.

Summary

In this chapter we have explored the Bridge design pattern. It provides convenient way for separating the abstraction from implementation through a bridge of abstract class or interface. The attached source code contains the example we discussed.

22. Interpreter Pattern

In this chapter I would like to demonstrate the usage of Interpreter pattern. This is not a widely used pattern but reveals a good way of programming. It says about creating a language along with an interpreter for the language. The language can be used to represent states, actions, loops etc. depending on the programming requirement. Here we are using the pattern to resolve a persistence problem.

Challenge

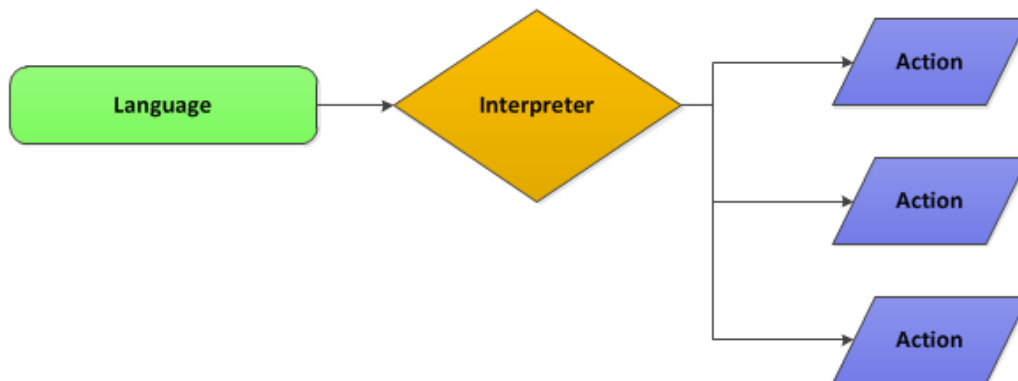
You are working on a Drawing application which allows creating lines and circles on a canvas. The classes Line and Circle do the job for you. You need to communicate the drawing with multiple users who have their own copy of the application. The current persistence of drawing using serialization of the above classes is creating large sized files and does not allow sending the message over a chat window. How to provide a better solution for persistence with smaller size and easier communication?

Definition

GoF Definition: "Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language"

Implementation

We can use the Interpreter pattern to approach the problem. We need to construct a language to include the commands and an interpreter to execute the actions.



Following are the classes involved for our drawing application:

- A Line Class
- A Circle Class

- An Interpreter Class

The Line and Circle class includes the drawing functionality. The Interpreter class takes care of converting the commands to invoking of appropriate line or circle classes.

The Language

The language part could be constituted by a series of sentences. The sentence could be as following:

line,10,10,10,10	Line command stating x, y, width, height
circle,10,10,50	Circle command stating x, y, radius

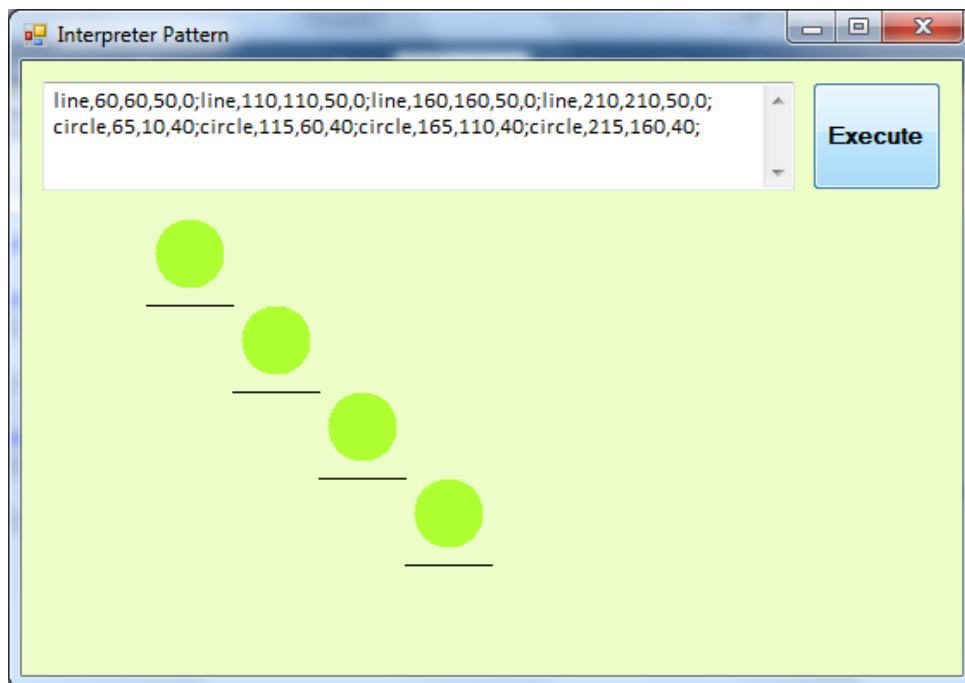
Please note that the arguments are separated using comma (,).

The sentence could be separated using semicolon (;) as shown below.

line,60,60,50,0;line,110,110,50,0;

Executing the Application

We can enter the series of commands separated by semicolon (;). The commands are entered in the application.



Clicking Execute you can see the above drawings of lines and circles.

The body of the Interpreter Execute() method is given below:

```
public void Execute(string commands, Graphics graphics)
{
    graphics.Clear(Color.FromArgb(237, 253, 200));

    int i = 0;

    foreach (string rawCommand in commands.Split(';'))
    {
        string command = rawCommand.Replace(Environment.NewLine,
string.Empty).Trim();

        if (command.StartsWith("line"))
        {
            i = 0;
            int x = 0, y = 0, width = 0, height = 0;

            foreach (string argument in command.Split(','))
            {
                if (i == 1)
                    x = int.Parse(argument);

                else if (i == 2)
                    y = int.Parse(argument);

                else if (i == 3)
                    width = int.Parse(argument);

                else if (i == 4)
                    height = int.Parse(argument);

                i++;
            }

            new Line(graphics).Draw(x, y, width, height);
        }
        else if (command.StartsWith("circle"))
        {
            i = 0;
            int x = 0, y = 0, radius = 0;

            foreach (string argument in command.Split(','))
            {
                if (i == 1)
                    x = int.Parse(argument);

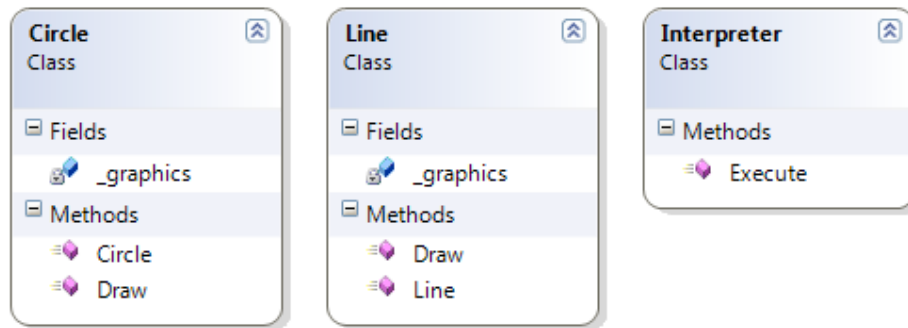
                else if (i == 2)
                    y = int.Parse(argument);

                else if (i == 3)
                    radius = int.Parse(argument);

                i++;
            }

            new Circle(graphics).Draw(x, y, radius);
        }
    }
}
```

Following is the class diagram:



This concludes our application using the Interpreter Pattern. Now the actions could be converted to a language for interpreting later.

Comparison of Command Pattern and Interpreter Pattern

You can find some similarities between the Command and Interpreter patterns. The command pattern says about converting the actions into command for logging or undo purposes. Here the commands are objects but for the Interpreter pattern the commands are sentences.

The Command pattern can be used for persisting objects by serializing the command list and results in large sized files compared with Interpreter pattern. The Interpreter pattern provides an easier approach in the runtime but comes at the cost of developing and interpreter.

Applications of Interpreter Pattern

You are creating your own Photoshop application. The objects drawn on the layer can be converted to a series of commands. Later the entire drawing could be represented using the command file which will be small in size compared to the entire canvas serialized. You will get the advantage of sending the file to another person where he/she can:

- Regenerate the Drawing
- View all the Draw Actions
- Modify the Draw Action Items
- Undo the last draw actions

References

http://www.dofactory.com/Patterns/PatternInterpreter.aspx#_self1

Summary

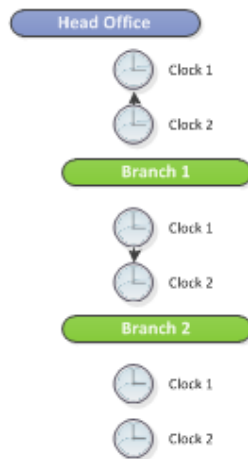
In this chapter we have seen about the Interpreter pattern with a small application using it. The above application shows only a simple usage for learning purpose but in the real world scenarios the language may include loops, conditions etc. The attachment contains the source code of application we have discussed.

23. Composite Pattern

In this chapter we can explore the Composite Design Pattern. It provides a better approach in addressing communication with individual and group of objects. As usual we can start with the Challenge – Solution approach.

Challenge

You are working on a distributed application containing Head Office and Branch Offices. Each office will be having 2 Clock Controls which needed to be updated globally during Daylight Saving times.



The above diagram shows the current structure. There will be only 1 Head Office and multiple branches under it. The head office and each branch will be having 2 clock controls. The time changing process starts from the head office and in the current approach, the head office has to remember all branches and all clocks inside each branch. This is very tedious and will break when there is consolidation of branches under one branch group.

How to provide a better solution?

Definition

GoF Definition: "Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly"

Implementation

We can use the Composite Design Pattern in the above scenario. Once our solution is implemented the Head Office needs to think about only the Branches. The Branches will think about their respective Clocks.

There are 2 classes in our solution: Office and Clock. The Office class can represent Head Office and Branches. The Clock class manages the clock control for changing time.

We need to introduce one interface which will be implemented by the Office and Clock classes.

```
interface IComponent
{
    void Add (IComponent noti fier);
    void SetTime(DateTime time);
}
```

The Add() method ensures adding a component of type IComponent. It can be used to add an Office or Clock instance.

The SetTime() method can be used to change the current time.

Class Implementations

Following are the implementations of the Office and Clock classes:

```
class Office : IComponent
{
    private IList<IComponent> _list = new List<IComponent>();

    public void Add(IComponent noti fier)
    {
        _list.Add(noti fier);
    }

    public void SetTime(DateTime time)
    {
        foreach (IComponent n in _list)
            n.SetTime(time);
    }
}

class Clock : IComponent
{
    public ClockControl ClockControl;

    public void Add(IComponent noti fier)
    {
        throw new ApplicationException("You cannot add IClock!");
    }

    public void SetTime(DateTime time)
    {
        ClockControl.CurrentTime = time;
    }
}
```

Please note that the SetTime() implementation of Office class takes care of informing all the added IComponent instances. Thus the Head Office can notify all the branches. The branches will notify all the clock instances.

Creating Instances

Following is the code of instance creation for Head Office and Branches.

```
_headOffice = new Office();
_headOffice.Add(new Clock() { ClockControl = hForm.clock1 });
_headOffice.Add(new Clock() { ClockControl = hForm.clock2 });

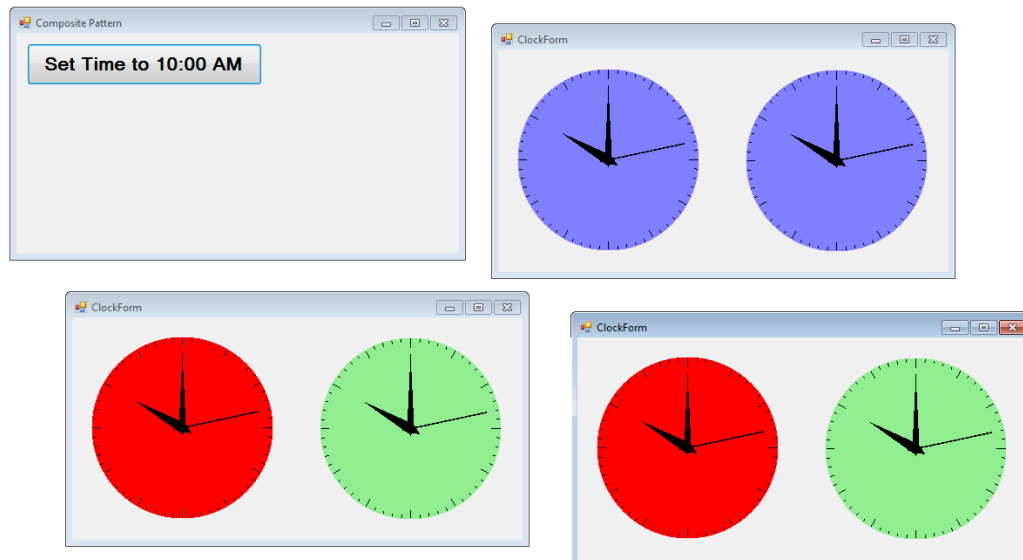
Office branch1 = new Office();
branch1.Add(new Clock() { ClockControl = b1Form.clock1 });
branch1.Add(new Clock() { ClockControl = b1Form.clock2 });
_headOffice.Add(branch1);

Office branch2 = new Office();
branch2.Add(new Clock() { ClockControl = b2Form.clock1 });
branch2.Add(new Clock() { ClockControl = b2Form.clock2 });
_headOffice.Add(branch2);
```

Please note that the branch instances are added to the head office.

Running the Application

On clicking the Set Time button you can see all the clocks are reset to 10:00 AM.



Here the Head Office does not need to think about all the clock instances of branches. Here the individual objects (clocks) and composite objects (branches) are treated uniformly satisfying the pattern definition.

Summary

In this chapter we have explored the Composite Pattern. It provides a centralized approach in talking with individual and combined objects. The attachment contains the example application including the clock control we have discussed.

References

The ebook is compiled on information collected from Books, MSDN and other Online Resources. It took almost 1 year to cover all the chapters and so some of the reference links will be obsolete.

You can find more information on Design Patterns using the following links:

<http://www.dofactory.com>

<http://www.oodeesign.com>

Source Code

The source code for the ebook can be found over <http://jeanpaulva.com>.

Conclusion

I hope information provided in this book was useful. There are more patterns additional to the 23 Design Patterns revealed in this book. I would like to cover them as well in future.

I would like to know the feedback on the technical content and writing style so that I can improve in my further books.

You can contact me over:

jeanpaulva@gmail.com
jeanpaulva@hotmail.com

Twitter: twitter.com/#!/jeanpaulva1

Website: <http://jeanpaulva.com/>