Design Patterns

From Wikipedia, the free encyclopedia

Design patterns as per GoF

Design Patterns: Elements of Reusable Object-Oriented Software (1994) is a software engineering book describing software design patterns. The book was written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, with a foreword by Grady Booch. The book is divided into two parts, with the first two chapters exploring the capabilities and pitfalls of object-oriented programming, and the remaining chapters describing 23 classic software design patterns. The book includes examples in C++ and Smalltalk.

It has been influential to the field of software engineering and is regarded as an important source for object-oriented design theory and practice. More than 500,000 copies have been sold in English and in 13 other languages. The authors are often referred to as the Gang of Four (GoF).

Creational patterns:

1. Abstract factory: Provides one level of interface higher than the factory pattern. It is used to return one of several factories.
2. Builder: Construct a complex object from simple objects step by step.
3. Factory method: Provides an abstraction or an interface and let's subclass or implementing classes decide which class or method should be instantiated or called, based on the conditions or parameters given.
4. Prototype: Cloning an object by reducing the cost of creation.
5. Singleton: One instance of a class or one value accessible globally in an application.

Structural patterns:

1. Adapter: Convert the existing interfaces to a new interface to achieve compatibility and reusability of the unrelated classes in one application. Also known as Wrapper pattern.
2. Bridge: Decouple an abstraction or interface from its implementation so that the two can vary independently
3. Composite: Build a complex object out of elemental objects and itself like a tree structure.
4. Decorator: Attach additional responsibilities or functions to an object dynamically or statically. Also known as Wrapper.
5. Façade: Make a complex system simpler by providing a unified or general interface, which is a higher layer to these subsystems.
6. Flyweight: Make instances of classes on the fly to improve performance efficiently, like individual characters or icons on the screen.
7. Proxy: Use a simple object to represent a complex one or provide a placeholder for another object to control access to it.

Behavioral patterns:

1. Chain of Responsibility: Let more than one object handle a request without their knowing each other. Pass the request to chained objects until it has been handled.
2. Command: Streamline objects by providing an interface to encapsulate a request and make the interface implemented by subclasses in order to parameterize the clients.
3. Interpreter: Provides a definition of a macro language or syntax and parsing into objects in a program.
4. Iterator: Provide a way to move through a list of collection or aggregated objects without knowing its internal representations.
5. Meditator: Define an object that encapsulates details and other objects interact with such object. The relationships are loosely decoupled.
6. Memento: To record an object internal state without violating encapsulation and reclaim it later without knowledge of the original object.
7. Observer: One object changes state, all of its dependents are updated automatically.
8. State: An object's behavior change is represented by its member classes, which share the same super class.
9. Strategy: Group several algorithms in a single module to provide alternatives. Also known as policy.
10. Template Method: Provide an abstract definition for a method or a class and redefine its behavior later or on the fly without changing its structure.
11. Visitor: Define a new operation to deal with the classes of the elements without changing their structures.

Definition

- Design patterns are documented tried and tested solutions for recurring problems in a given context
- Design pattern is not a silver bullet.
- Do not over do design patterns

Sorting

Bubble Sort

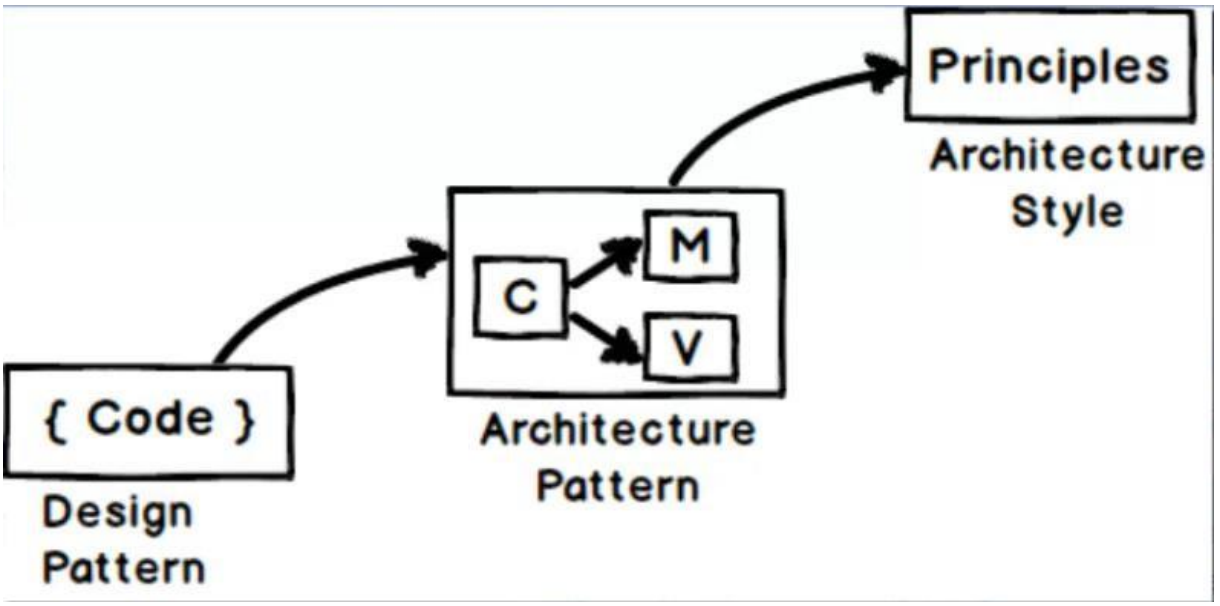
you have a problem context and the proposed solution

Creational Patterns

Structural Patterns

Behavioral Patterns

Design Pattern
VS
Architecture Pattern
VS
Architecture Style

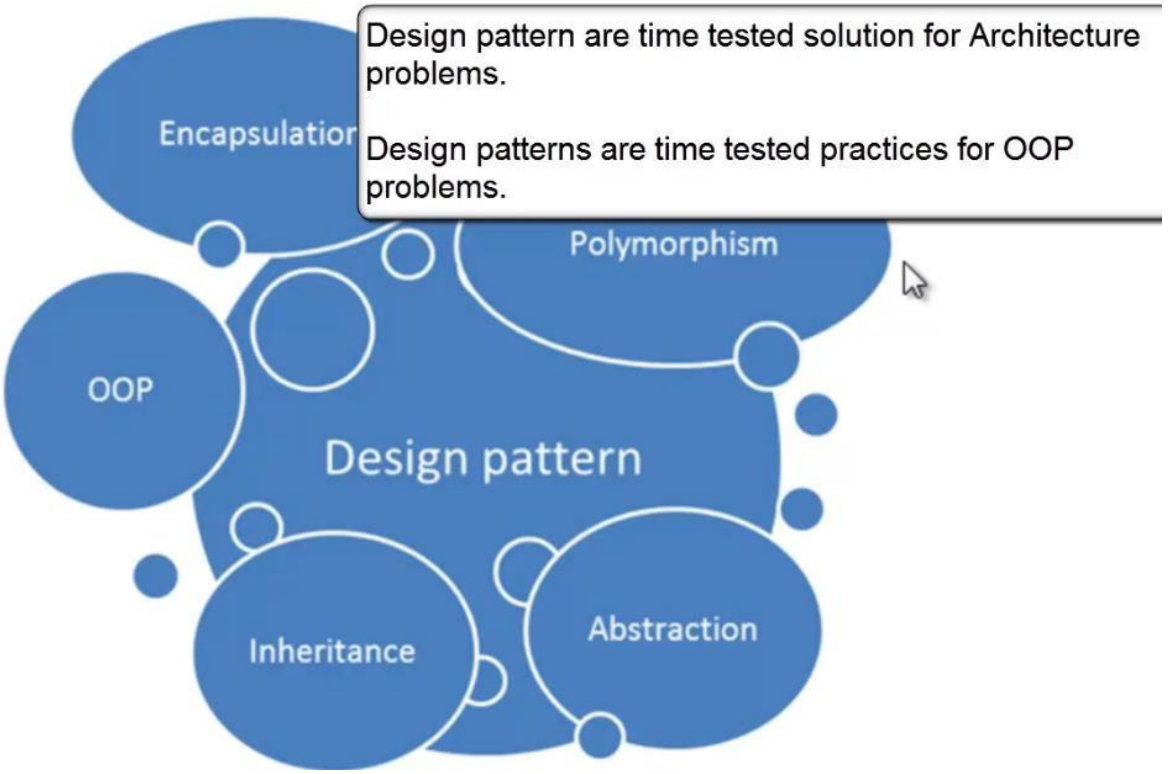


SUDO CODE IS DESIGN PATTERN

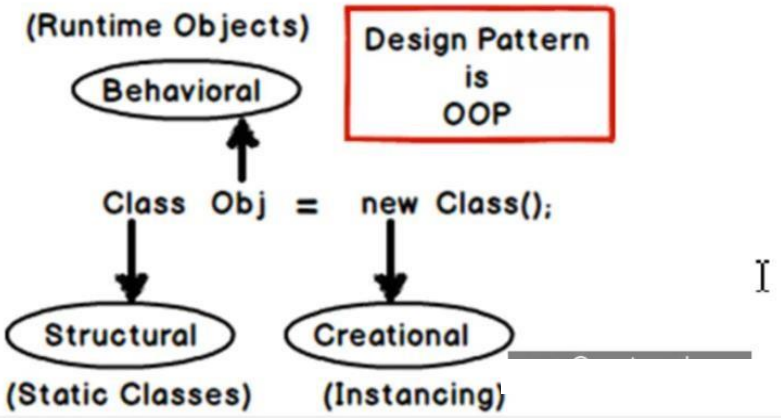
REST IS A ARCHITECTURAL SYTLE WICH FOLLOW HTTP PROTOCOL ,

Below are some examples for each one of them.

Design pattern	Factory , Iterator , Singleton
Architecture pattern	MVC,MVP,MVVM
Architecture style	REST,SOA , IOC



OOP Phase	Design pattern category
Template / Class creation problem	Structural design pattern.
Instantiation problems	Creational design pattern.
Runtime problems	Behavioral design pattern.



Factory Pattern

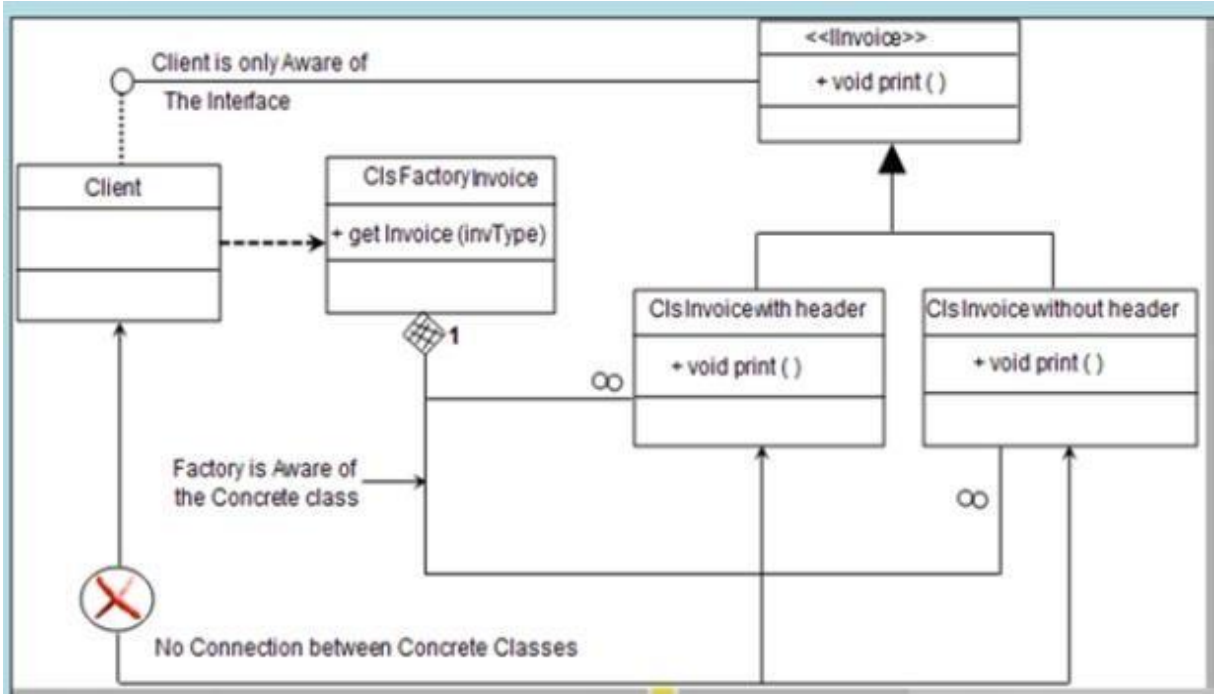
•It's a type of creational pattern.

•You can make out from the name factory itself it's meant to construct and create something.

```
if (intInvoiceType == 1)
{
   objinv = new clsInvoiceWithHeader();
}
else if (intInvoiceType == 2)
{
    objinv = new clsInvoiceWithOutHeaders();
}
```

Problems with code

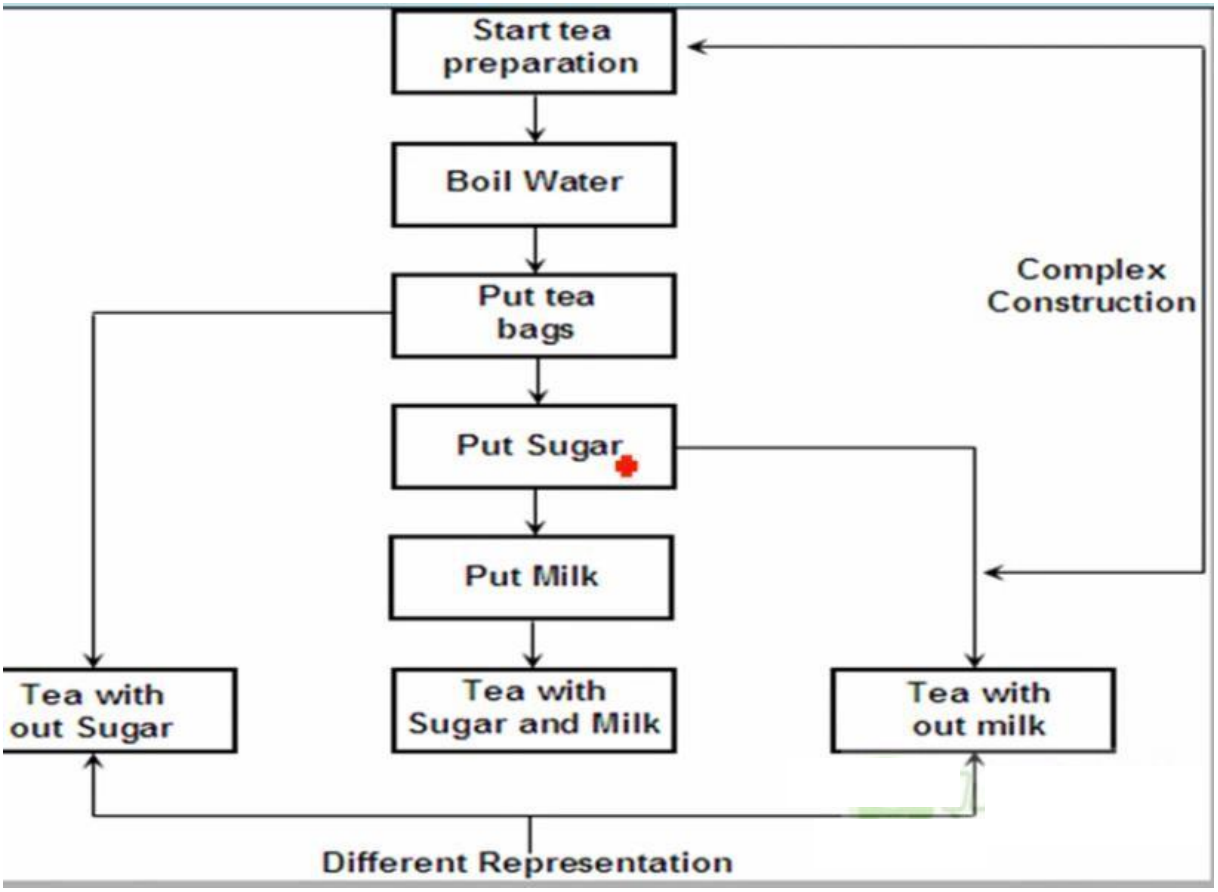
- Lot of Scattered new Keyword
- Client is aware of all invoice types



Builder Patterns

Definition

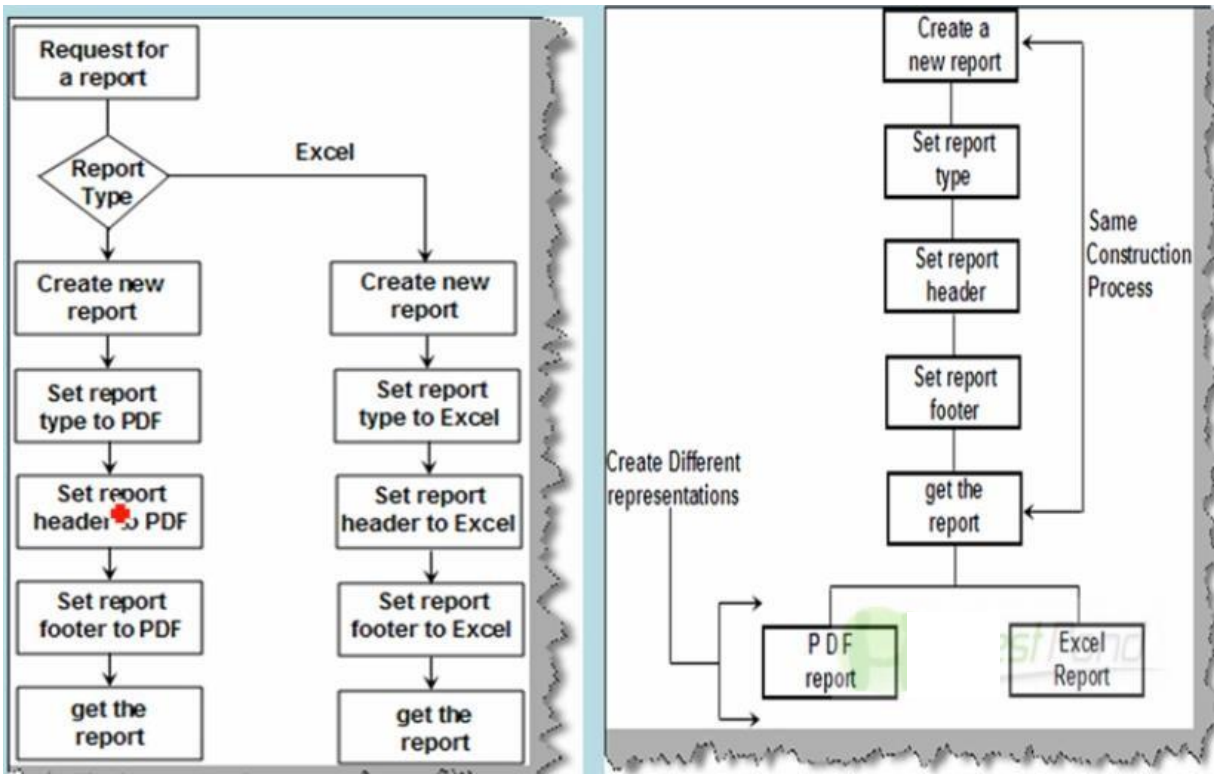
- They fall in to creational categories.
- Builder pattern helps us to separate the construction of a complex object from its representation so that the same construction process can create different representations
- Builder pattern is useful when the construction of the object is very complex.



Builder: - Builder is responsible for defining the construction process for individual parts. Builder has those individual processes to initialize and configure the product.

Director: - Director takes those individual processes from the builder and defines the sequence to build the product.

Product: - Product is the final object which is produced from the builder and director coordination.




```
clsReport objReport;
clsDirector objDirector = new clsDirector();

ReportPDF objRptPdf = new ReportPDF();
objReport = objDirector.MakeReport(objRptPdf);
objReport.displayReport();

ReportExcel objRptExcel = new ReportExcel();
objReport = objDirector.MakeReport(objRptExcel);
objReport.displayReport();
```

Client code

```
public class clsDirector
{
    public clsReport MakeReport(ReportBuilder ob
    {
        objBuilder.createNewReport();
        objBuilder.setReportType();
        objBuilder.setReportHeader();
        objBuilder.setReportFooter();
        return objBuilder.getReport();
    }
}
```

Director code

```
public class ReportPDF : ReportBuilder
{
    public override void setReportType()
    {
        objReport.strReportType = "PDF";
    }
    public override void setReportHeader()
    {
        objReport.setReportHeader(" XYZ PVT LTD PDF rep
    }
    public override void setReportFooter()
    {
        objReport.setReportFooter("End of PDF report");
    }
}
```

PDF builder

```
public class ReportExcel : ReportBuilder
{
    public override void setReportType()
    {
        objReport.strReportType = "EXCEL";
    }
    public override void setReportHeader()
    {
        objReport.setReportHeader(" XYZ PVT LTD EXCEL re
    }
    public override void setReportFooter()
    {
        objReport.setReportFooter("End of EXCEL report").
    }
```

Excel builder

Decoupling builder process from construction process.

Prototype Patterns

Definition

- They fall in to creational categories.
- It gives us a way to create new objects from the existing instance of the object. In one sentence we clone the existing object with its data. By cloning any changes to the cloned object does not affect the original object

```
Class1 obj1 = new Class1();
```
→ Object 1 created

```
Class1 obj2 = new Class1();
```
→ Object 2 created

```
obj1.strCode = "Old value";
```
→ Object 1 value set

```
obj2 = obj1;
```
→ Object 1 set to Object 2

```
obj2.strCode = "New Value";
```
→ Object 2 value set but that changesthe value of object 1

Final Output →


```
MessageBox.Show("Value of object1 = " + obj1.strCode + "\n value of object2 =" + obj2.strCode);
```

```
public class clsCustomer
{
    public string strCustomerCode = "";
    public string strCustomerName = "";
    public clsCustomer getClone()
    {
        return (clsCustomer) this.MemberwiseClone();
    }
}
```

Class to be cloned

'MembersWiseClone' function creates a complete new copy of the object

client creates the object of the clsCustomer class

```
clsCustomer obj1 = new clsCustomer();

obj1.strCustomerCode = "001";
obj1.strCustomerName = "OldCustomer";

clsCustomer obj2;
obj2 = obj1.getClone();
```

Client

The first copy of the object is created

Values are set for the first object

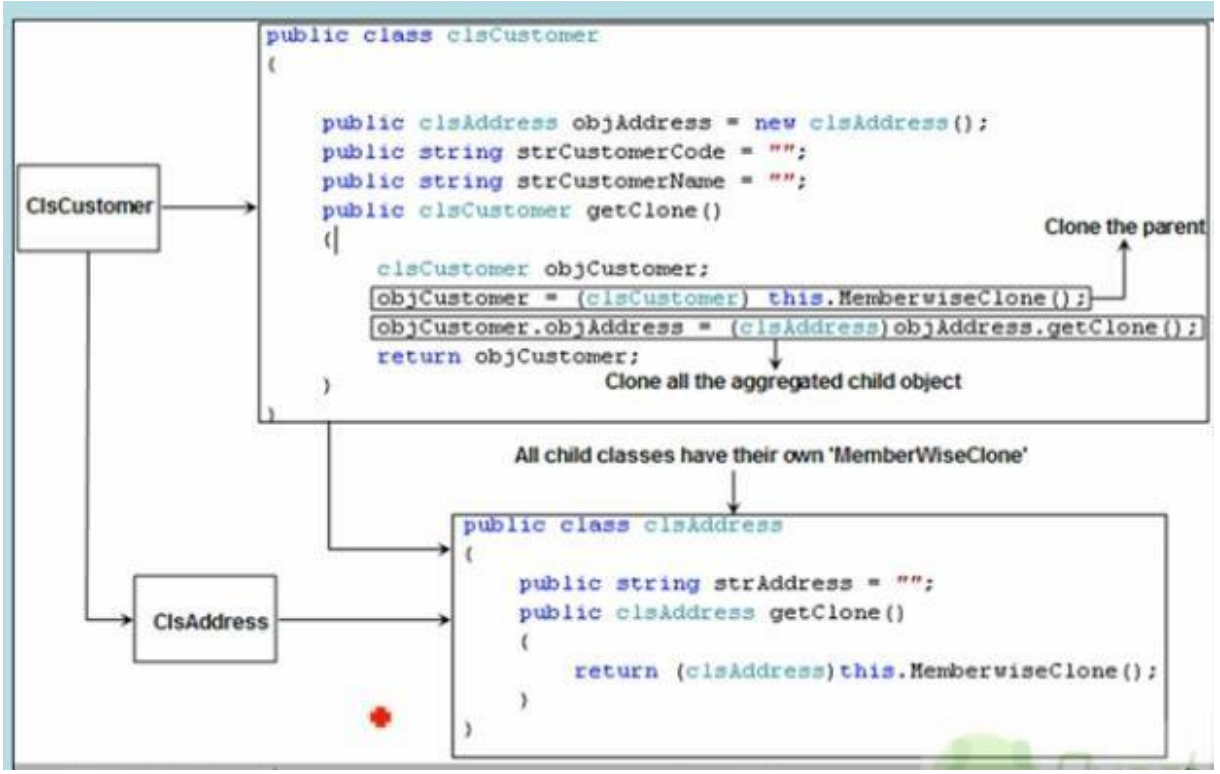
Second copy of the object is created

Call the getClone to create a copy of the object

In C# we use the 'MemberWiseClone' function while in JAVA we have the 'Clone' function to achieve the same.

Shallow cloning and deep cloning

- When the parent objects are cloned with their containing objects it's called as *deep cloning* and when only the parent is clones its termed as *shallow cloning*.



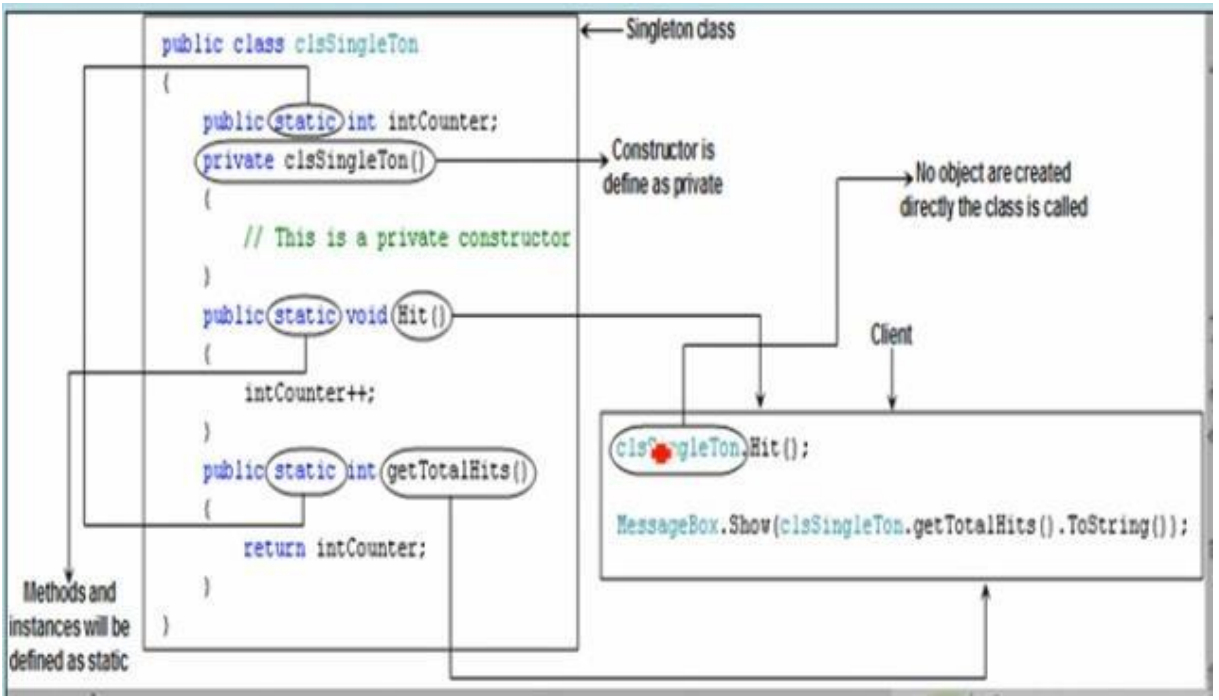
Singleton Pattern

Fundamentals

- They fall in to creational categories.
- There are situations in a project where we want only one instance of the object to be created and shared between the clients.
- No client can create an instance of the object from outside.

Steps to implement singleton patterns

- Define the constructor as private
- Define the instances and methods as static.

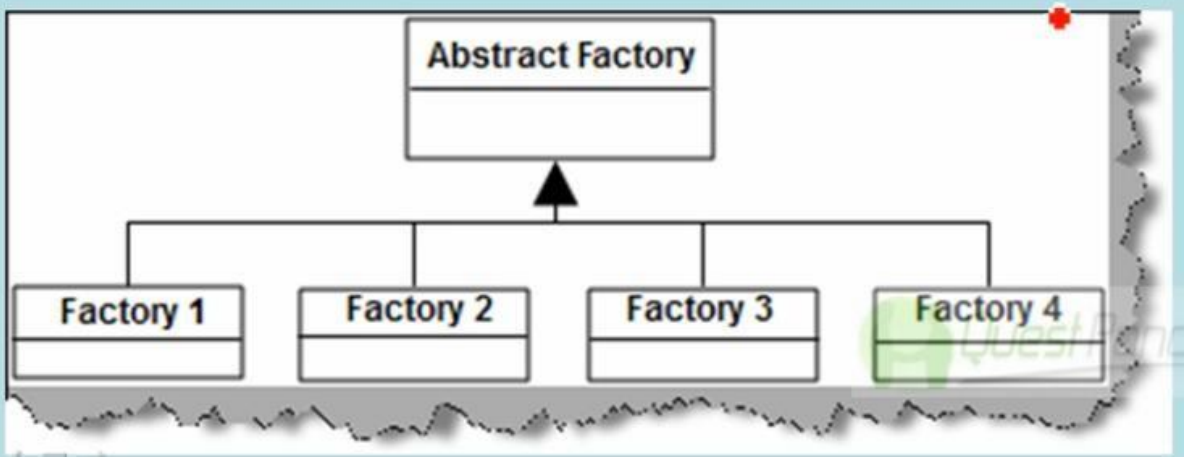


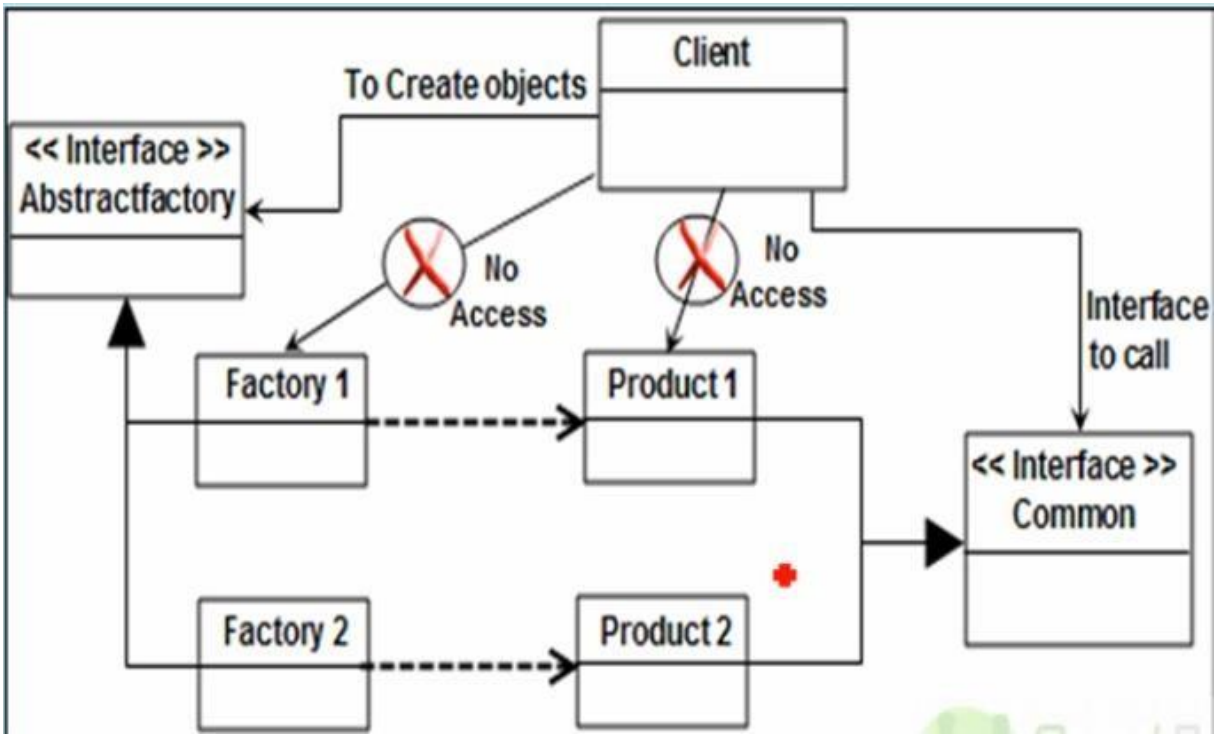
Abstract Factory Pattern

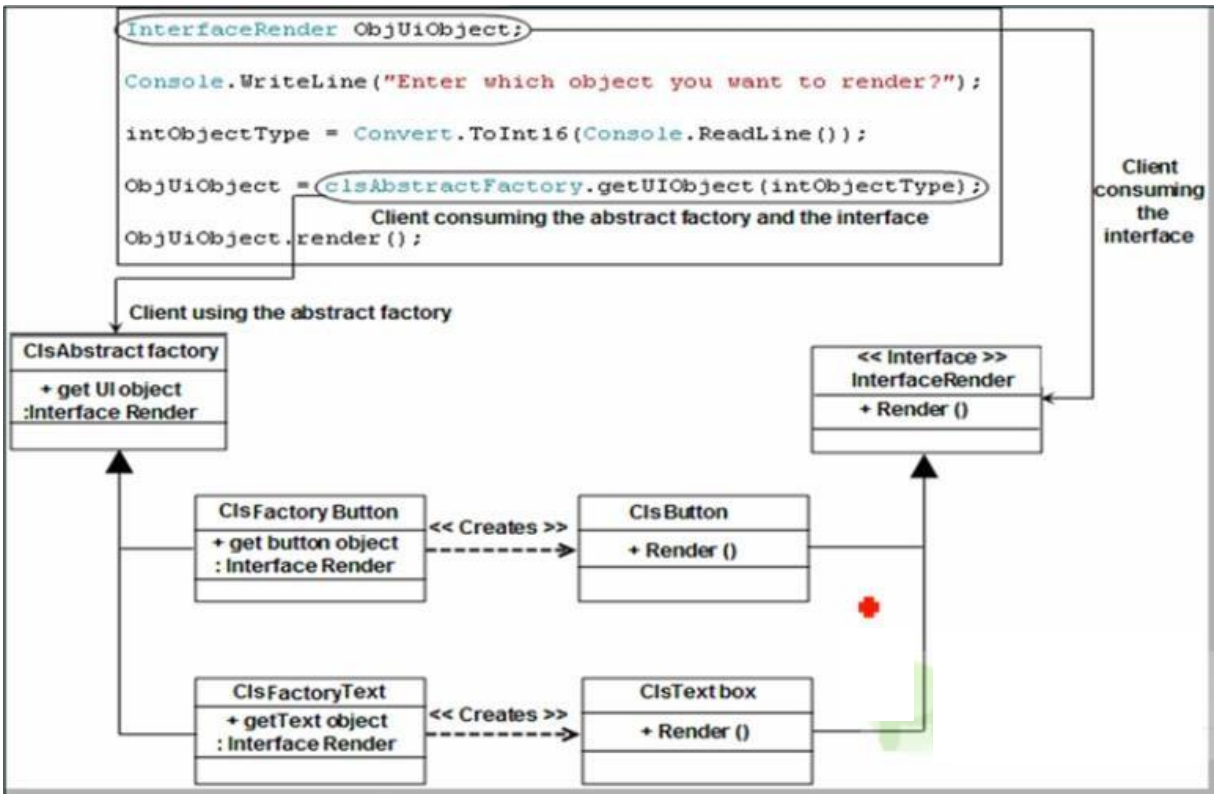
Abstract Factory Pattern

- Abstract factory expands on the basic factory pattern.
- Abstract factory helps us to unite similar factory pattern classes in to one unified interface.

A factory class helps us to centralize the creation c
classes and types. Abstract factory helps us to
bring uniformity between related factory patterns
which leads more simplified interface for the client.






InterfaceRender ObjUiObject;
Console.WriteLine("Enter which object you want to render?");
intObjectType = Convert.ToInt16(Console.ReadLine());
ObjUiObject = clsAbstractFactory.getUIObject(intObjectType);
Client consuming the abstract factory and the interface
ObjUiObject.render();
Client consuming the interface
Client using the abstract factory
ClsAbstract factory
+ get UI object
:Interface Render
<< Interface >>
InterfaceRender
+ Render ()
ClsFactory Button
+ get button object
: Interface Render
<< Creates >>
ClsButton
+ Render ()
ClsFactoryText
+ getText object
: Interface Render
<< Creates >>
ClsText box
+ Render ()

Structural Patterns

ADAPTER

DECORATOR

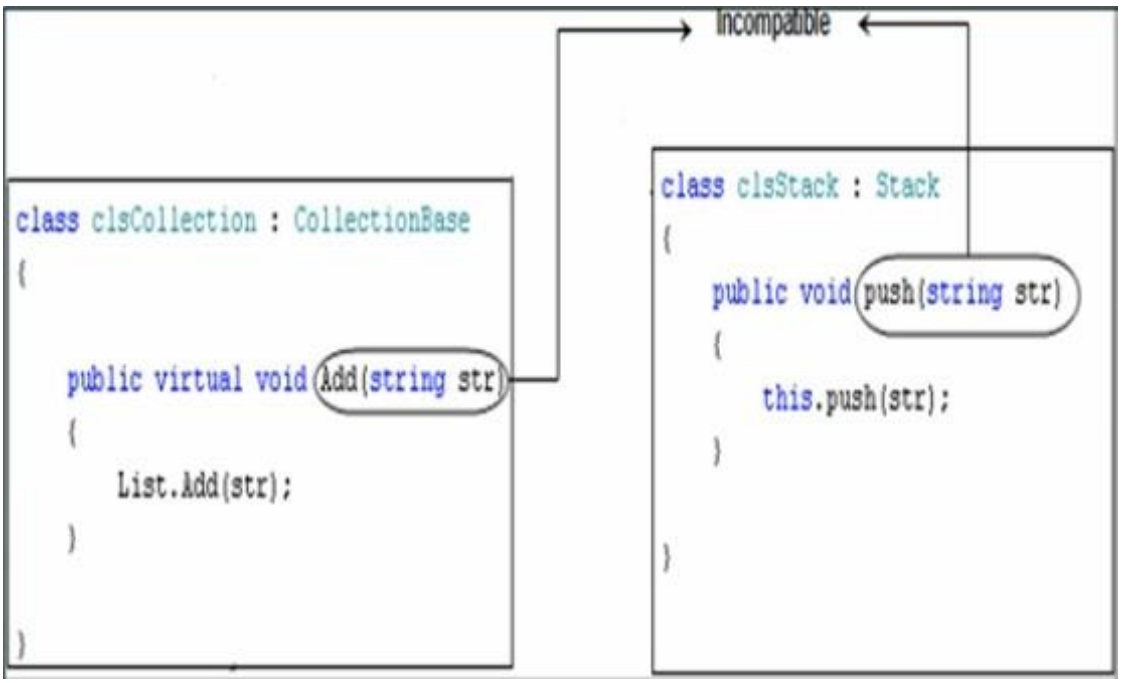
BRIDGE

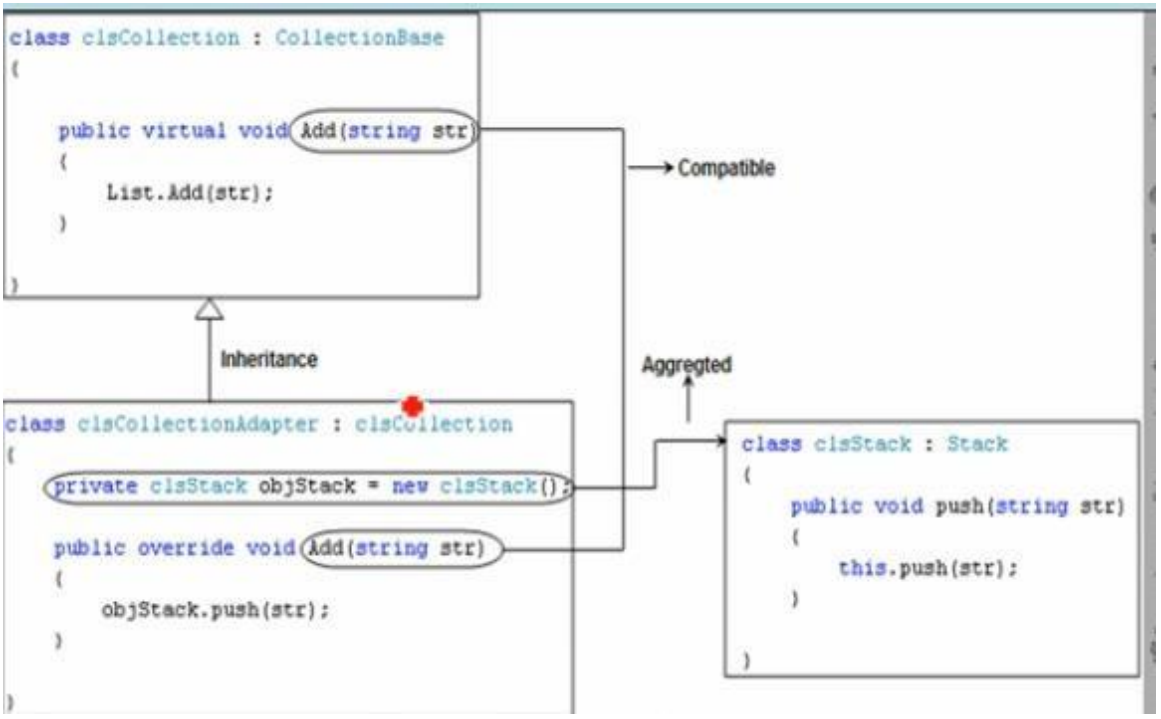
FAÇADE

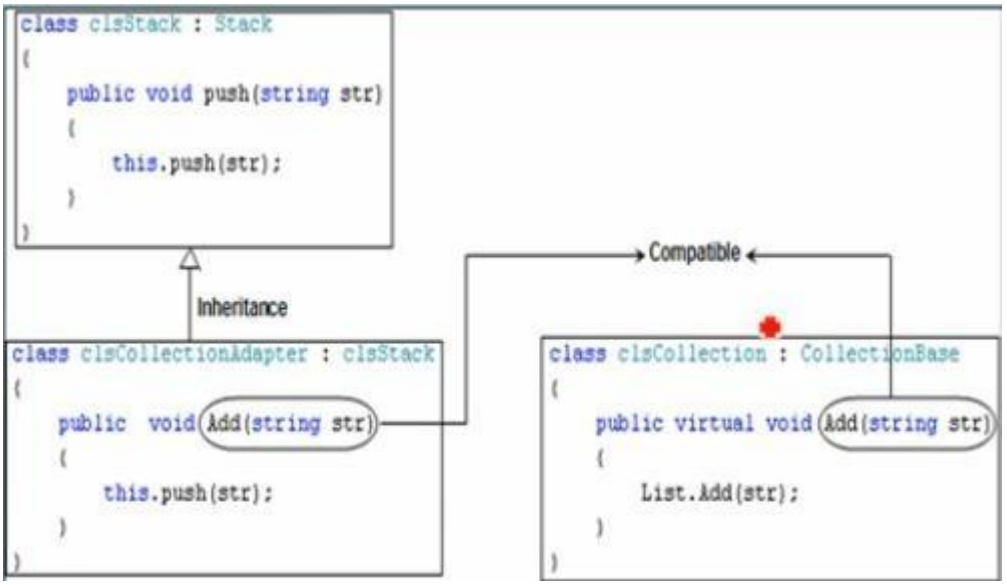
Adapter Pattern

Fundamentals

- They fall in to structural pattern categories.
- Many times two classes are incompatible because of incompatible interfaces.
- Adapter helps us to wrap a class around the existing class and make the classes compatible with each other.







Decorator Pattern

Fundamentals

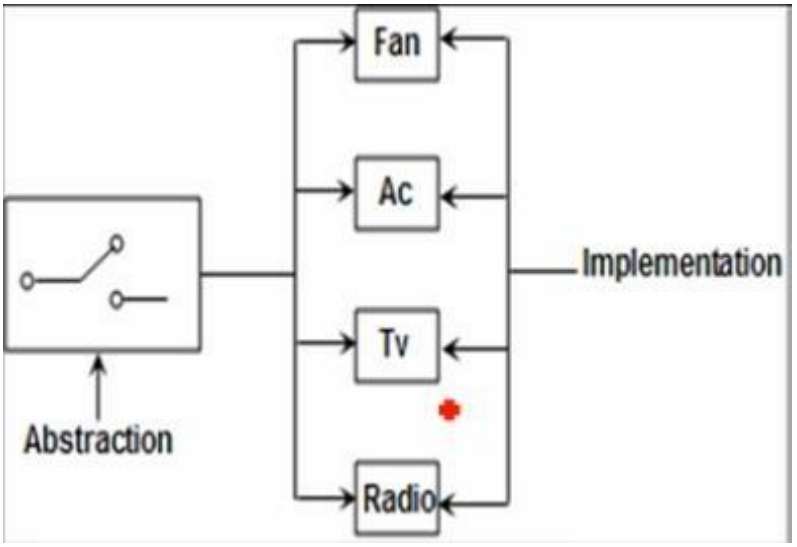
- They fall in to structural pattern categories.
- Decorator pattern allows creating inherited classes which are sum of all part of the parent.

```
class Class1
{
    public void someFunction()←——Some function
    {
        Console.WriteLine("Some function");
    }
}
class Class2 : Class1←——Inher it from the above class
{
    public void someMoreFunction()←——Add more function
    {
        Console.WriteLine("Some more function");
    }
}
```

Bridge Pattern

Fundamentals

- They fall in to structural pattern categories.
- Bridge pattern helps to decouple abstraction from implementation.
- So if the implementation changes it does not affect abstraction and vice versa



```
public interface IEquipment
{
    void Start();
    void Stop();
}
```

Inheritance

```
class clsRefrigerator : IEquipment
{
    public  void Start()
    {
        Console.WriteLine("Start Compressor");
        Console.WriteLine("Start Ice Cooling");
    }
    public  void Stop()
    {
        Console.WriteLine("Stop Ice Cooling");
        Console.WriteLine("Stop Compressor");
    }
}
```

```
class clsBulb : IEquipment
{
    public void Start()
    {
        Console.WriteLine("Warm up the bulb");
        Console.WriteLine("Glow the bulb");
    }
    public void Stop()
    {
        Console.WriteLine("Switch off the bulb");
    }
}
```

Implementation

```
interface ISwitch
{
    void On();
    void Off();
}
class Clsswitch : ISwitch
{
    private IEquipment objEquipment;
    public void setEquipMent(IEquipment Equipment)
    {
        objEquipment = Equipment;
    }
    public void On()
    {
        objEquipment.Start();

    }
    public void Off()
    {
        objEquipment.Stop();

    }
```

Abstraction interface

Aggregated Equipment object reference

Implement both the interface

Abstraction

```
clsBulb objBulb = new clsBulb();
clsRefrigerator objRefrig = new clsRefrigerator();
Clsswitch objSwitch = new Clsswitch();

objSwitch.setEquipMent(objBulb);
objSwitch.On();
objSwitch.Off();

objSwitch.setEquipMent(objRefrig);
objSwitch.On();
objSwitch.Off();
Console.ReadLine();
```

Create the implementation objects

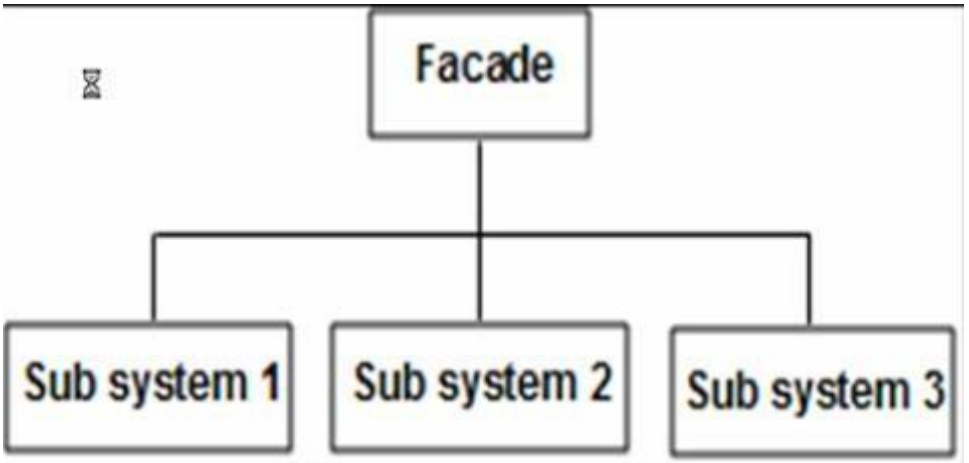
Abstraction objects

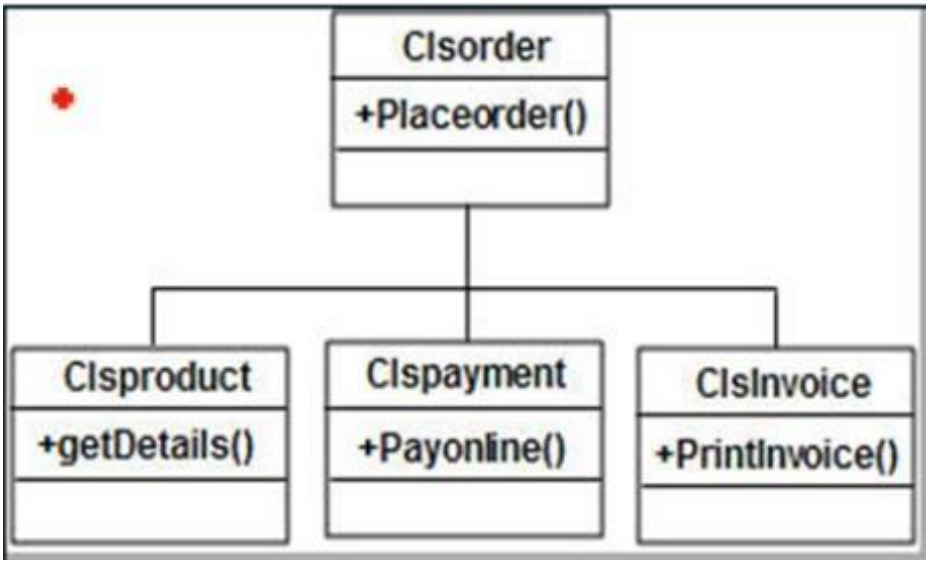
Implementation is seperate from abstraction

Facade Pattern

Fundamentals

- They fall in to structural pattern categories.
- Façade pattern sits on the top of group of subsystems and allows them to communicate in a unified manner.





8.

```
class clsProduct
{
    public string getDetails()
    {
        return "Product information";
    }
}
```

```
ass clsOrder

 public void PlaceOrder()
 {
     clsProduct objProduct = new clsProduct();
    clsPayment objPayment = new clsPayment();
    clsInvoice objInvoice = new clsInvoice();

     objProduct.getDetails();
     objPayment.PayOnline();
     objInvoice.PrintInvoice();
 }
```

```
class clsPayment
{
    public void PayOnline()
    {
    }
}
```

```
class clsInvoice
{
    public void PrintInvoice()
    {

    }
}
```

Behavioural Patterns

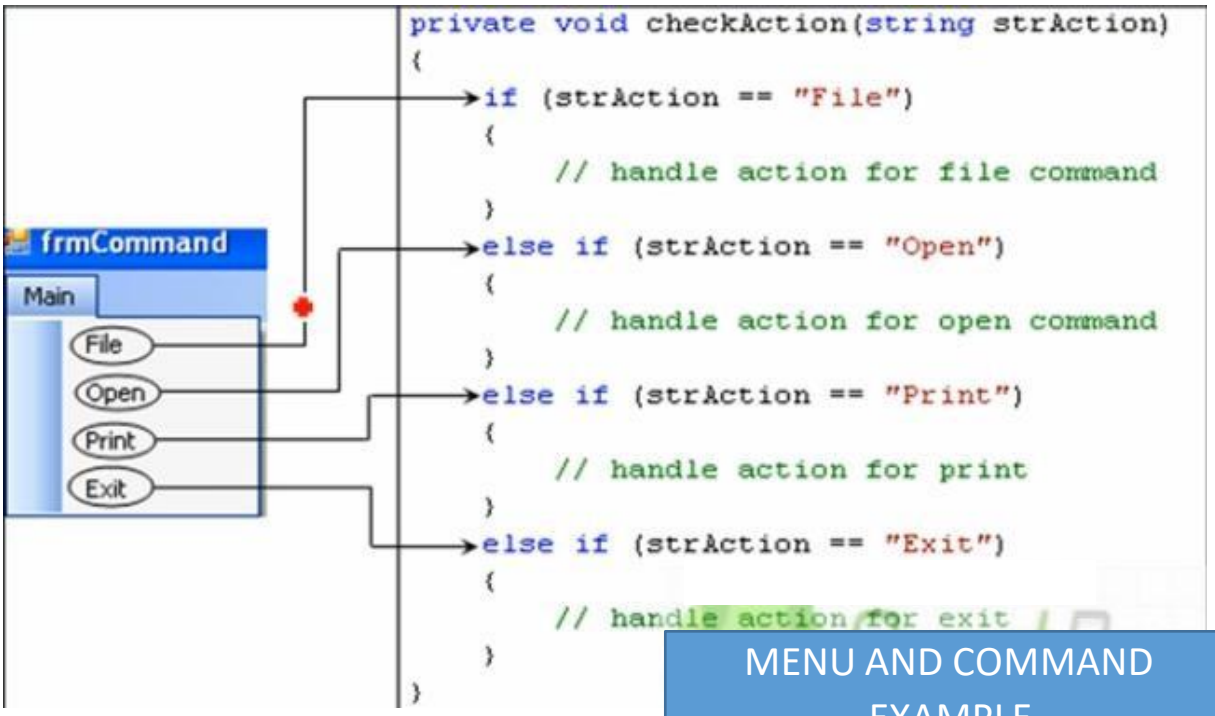
Behaviour category of a project are category of Design pattern, which help us to change behaviour of a project without altering the main structure.

1. Command
2. Mediator
3. Iterator
4. Observer
5. Strategy
6. State

Command Pattern

Fundamentals

- They fall in to behavioral categories.
- Command pattern allows a request to exist as an object.



MENU AND COMMAND EXAMPLE

Commands in Classes

```
class clsExecuteExit : IExecute
{
    public clsExecuteExit()
    {
        strCommand = "Exit";
    }
    public override void execute()
    {
        MessageBox.Show("Exit command");
    }
}
```

```
class clsExecuteOpen : IExecute
{
    public clsExecuteOpen()
    {
        strCommand = "Open";
    }
    public override void execute()
    {
        MessageBox.Show("Open command");
    }
}
```

```
abstract class IExecute
{
    public string strCommand="";
    public virtual void execute()
    {

    }
```

```
class clsExecutePrint : IExecute
{
    public clsExecutePrint()
    {
        strCommand = "Print";
    }
    public override void execute()
    {
        MessageBox.Show("Print command");
    }
```

```
class clsExecuteFile : IExecute
{
    public clsExecuteFile()
    {
        strCommand = "File";

    public override void execute()
    {
```

Invoker executes the commands

```
class clsInvoker
{
    private ArrayList objArrayList = new ArrayList();
    public clsInvoker()
    {
        objArrayList.Add(new clsExecuteExit());
        objArrayList.Add(new clsExecuteFile());
        objArrayList.Add(new clsExecuteOpen());
        objArrayList.Add(new clsExecutePrint());
    }
    public IExecute getCommand(string strCommand)
    {
        foreach (object obj in objArrayList)
        {
            IExecute objExecute = (IExecute)obj;
            if (objExecute.strCommand == strCommand)
            {
                return objExecute;
            }
        }
        return null;
```

Add the command objects in a collection

Client

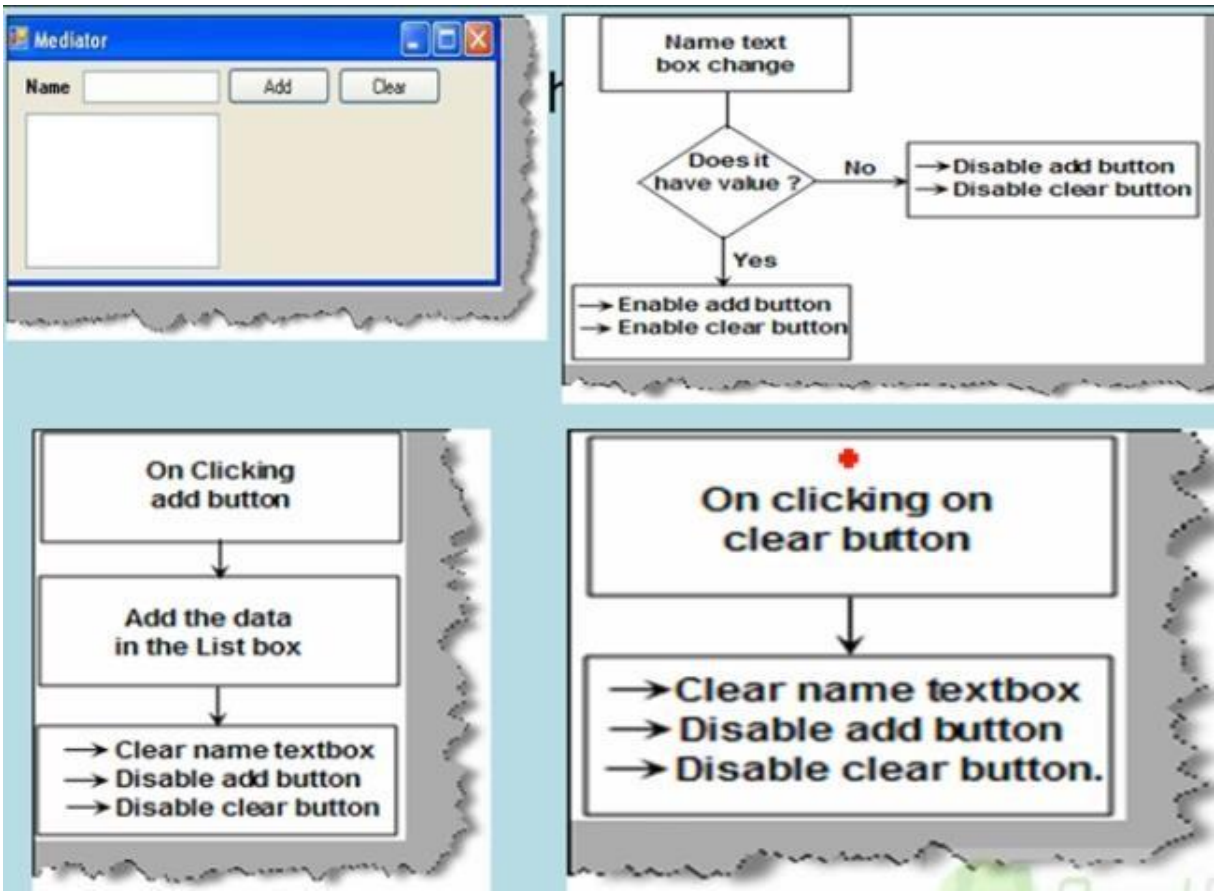
```
clsInvoker objInvoker = new clsInvoker();
IExecute iobjExecute= objInvoker.getCommand(strAction);
iobjExecute.execute();
```

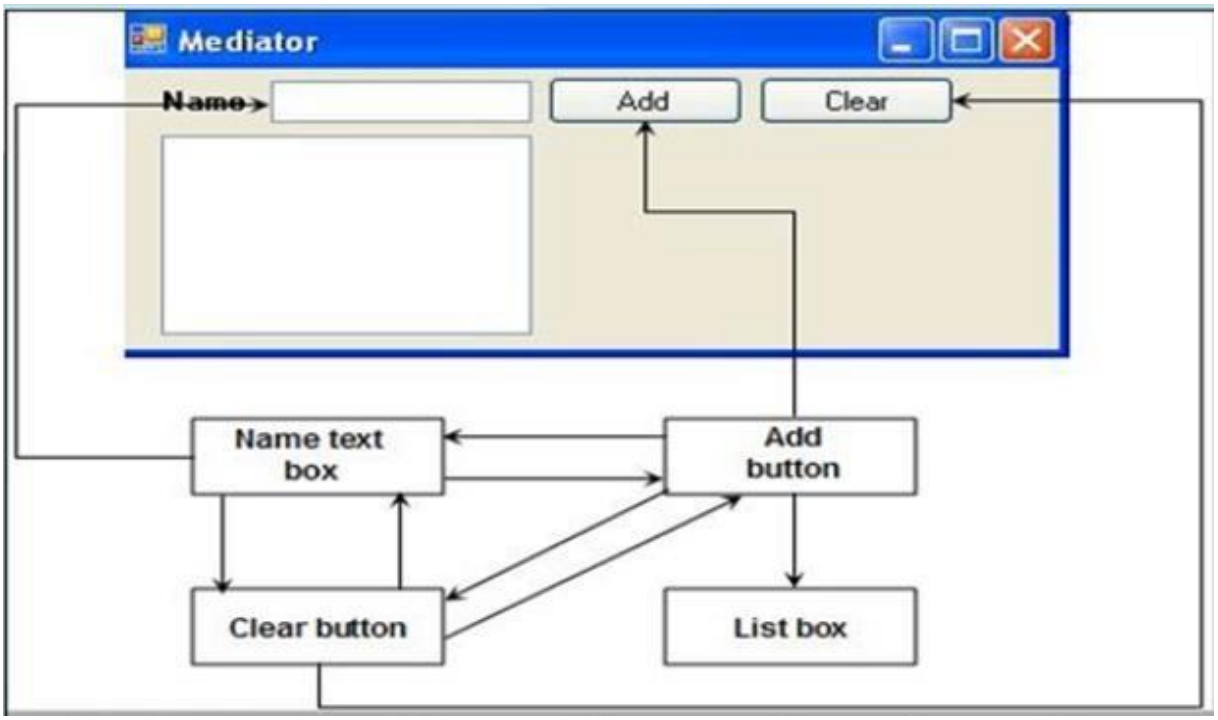
Depending on string value get the command object

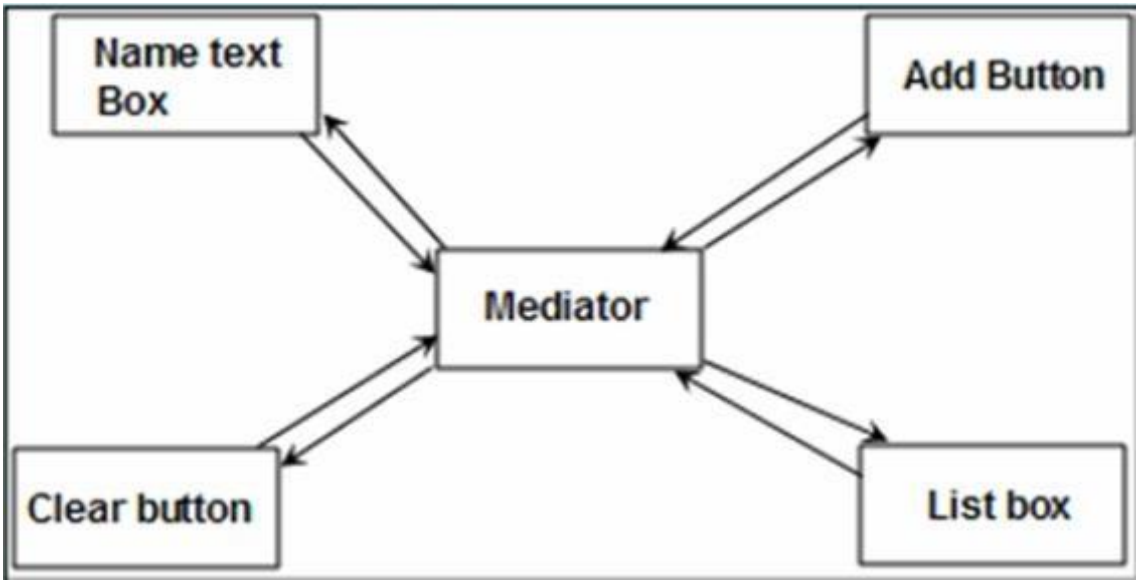
Mediator Pattern

Fundamentals

- They fall in to behavioral categories.
- Many a times in projects communication between components are complex.
- Due to this the logic between the components becomes very complex
- Mediator pattern helps the objects to communicate in a disassociated manner, which leads to minimizing complexity.




Mediator
Name
Add
Clear
Name text box
Add button
Clear button
List box


Name text Box
Add Button
Mediator
Clear button
List box

```
public void ClickClearButton()
{   Scenario 3
    objTextBox.Text = "";
    btnAdd.Enabled = false;
    btnClear.Enabled = false;
}
```

```
public void ClickAddButton()
{
    objListBox.Items.Add(objTextBox.Text);
    btnClear.Enabled = false;
    btnAdd.Enabled = false;
    objTextBox.Text = "";
}
```
Scenario 2

```
public void TextChange()
{
    if (objTextBox.Text.Length > 0
    {
        btnClear.Enabled = true;
        btnAdd.Enabled = true;
    }
    else
    {
        btnAdd.Enabled = false;
        btnClear.Enabled = false;
    }
}
```
Scenario 1

All complex logic in centralized in the mediator

```
class clsMediator
{
    private TextBox objTextBox;
    private ListBox objListBox;
    private Button btnClear;
    private Button btnAdd;
    public void Register(TextBox obj)...
    public void Register(ListBox obj)...
    public void Register(Button obj)...
    public void TextChange()...
    public void ClickAddButton()...
    public void ClickClearButton()...
}
```

```
public void Register(Button obj)
{
    if (obj.Name == "btnAdd")
    {
        btnAdd = obj;
    }
    else
    {
        btnClear = obj;
    }
    obj.Enabled = false;
}
```

```
public void Register(TextBox obj)
{
    objTextBox = obj;
}
```

```
public void Register(ListBox obj)
{
    objListBox = obj;
}
```

```
private clsMediator objMediator = new clsMediator();
public Form1()
{
    InitializeComponent();
    objMediator.Register(txtName);
    objMediator.Register(btnAdd);
    objMediator.Register(btnClear);
    objMediator.Register(lstName);

}
private void txtName_TextChanged(object sender, EventArgs e)
{
    objMediator.TextChange();
}

private void btnAdd_Click(object sender, EventArgs e)
{
    objMediator.ClickAddButton();
}

private void btnClear_Click(object sender, EventArgs e)
{
    objMediator.ClickClearButton();
}
```

Create the object of mediator class

Register all the UI components in the mediator

Call the appropriate events in the mediator to handle the complex logic

Iterator Pattern

Fundamentals

- They fall in to behavioral categories.
- Iterator pattern allows sequential access of elements with out exposing the inside code

```
class clsIterator
{
    private int intCurrentIndex = 0;
    private ArrayList objArray = new ArrayList();
    public void FillObjects()
    {
        clsCustomer objCustomer = new clsCustomer();
        objCustomer.strCustomer = "Shiv pvt ltd";
        objCustomer.strCode = "Cust001";
        objArray.Add(objCustomer);
                                        Add data to collection
        objCustomer = new clsCustomer();
        objCustomer.strCustomer = "Raju pvt ltd";
        objCustomer.strCode = "Cust002";
        objArray.Add(objCustomer);
    }
    public clsCustomer getByIndex(int intIndex)
    {
        if ((intIndex >= objArray.Count) || (intIndex == -1))
        {
            throw new Exception("End or Begin");
        }
        intCurrentIndex = intIndex;
        return (clsCustomer)objArray[intCurrentIndex];
    }
    public clsCustomer Prev()
    public clsCustomer Next()
    public clsCustomer getFirst()
    public clsCustomer getLast()
}
```

```
class clsCustomer
{
    public string strCustomer = "";
    public string strCode = "";
}
```

```
public clsCustomer Prev()
{
    int intTemp = intCurrentIndex;
    intTemp--;
    return (clsCustomer)getByIndex(intTemp);
}
```

```
public clsCustomer Next()
{
    int intTemp = intCurrentIndex;
    intTemp++;
    return (clsCustomer)getByIndex(intTemp);
}
```

```
public clsCustomer getFirst()
{
    return getByIndex(0);
}
```

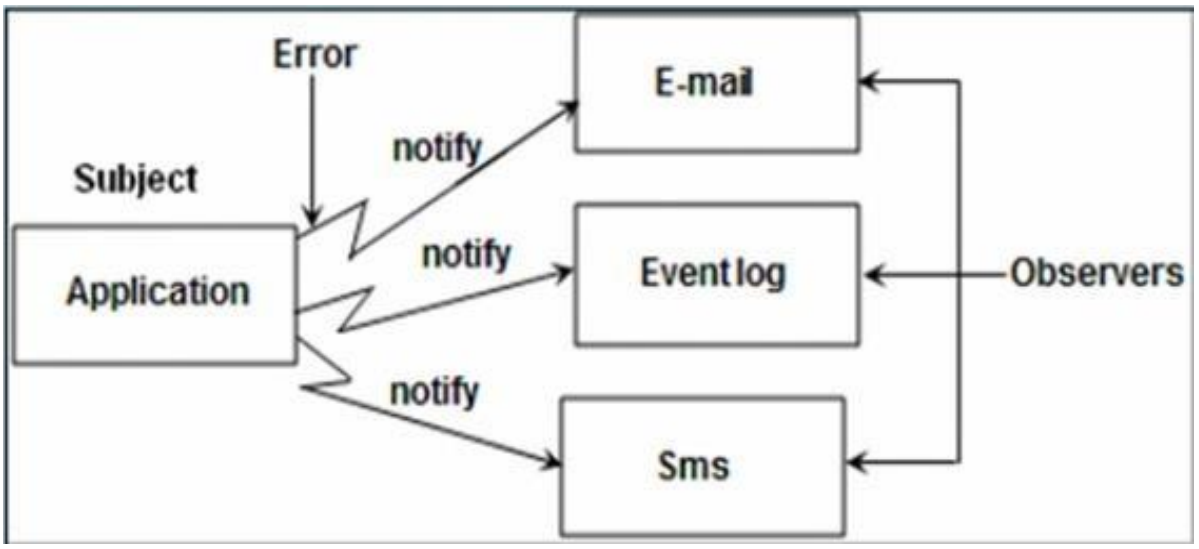
```
public clsCustomer getLast()
{
    return (clsCustomer)getByIndex(objArray.Count);
}
```

Get a object by index. if the index is out of bound then it throws a error

Observer Pattern

Fundamentals

- They fall in to behavioral categories.
- Observer pattern helps us to communicate between parent class and its associated or dependent classes.



```
public interface INotification
{
     void Notify();
}
public class clsEmailNotification : INotification
{
    public void Notify()
    {
        System.Console.WriteLine("Email notification code executed");
    }
}
public class clsEventNotifcation : INotification
{
    public void Notify()
    {
        System.Console.WriteLine("Event log notification code executed");
    }
}
public class clsSMSNotification : INotification
{
    public void Notify()
    {
        System.Console.WriteLine("SMS notification code executed");
    }
}
```

Interface

Inherited

```
public class clsNotifier
{
    private ArrayList objNotifications = new ArrayList();
    public void addNotification(INotification obj)
    {
        objNotifications.Add(obj);
    }
    public void removeNotification(INotification obj)
    {
        objNotifications.Remove(obj);
    }
    public void NotifyAll()
    {
        foreach (INotification objNotification in objNotifications)
        {
            objNotification.Notify();
        }
    }
}
```

Will contain all the notification objects

Add the subscribers i.e email,sms and eventlogs

Remove the notifications

Notify all the subscribers which are registered

```
// This application takes customer code and if
// the customer code length is above 20 it notifies
// the error to all the subscribers
string strCustomerCode = "";

// Notifier/Subject to notify all the observers
clsNotifier objNotifier = new clsNotifier();

// Add subjects/subscribers which needs to be notified
clsEmailNotification objEmailNotification = new clsEmailNotification();
clsEventNotifcation objEventNotification = new clsEventNotifcation();
objNotifier.addNotification(objEmailNotification);
objNotifier.addNotification(objEventNotification);

// create a error by entering length more than 10 characters
Console.WriteLine("Enter Customer Code");
strCustomerCode = Console.ReadLine();

// if the length is more than 10 characters notify all subjects/subscribers
if (strCustomerCode.Length > 10)
{
    objNotifier.NotifyAll();
}

Console.ReadLine();
```

Create a object of notifier

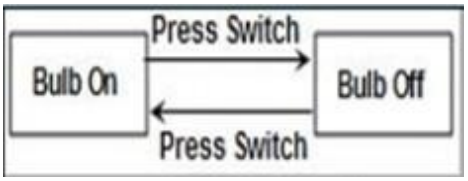
Create the object and add the subscribers to the notifiers

If the customer length is more than 20 characters then send notificationsto all thesubscribers/observers

State Pattern

Fundamentals

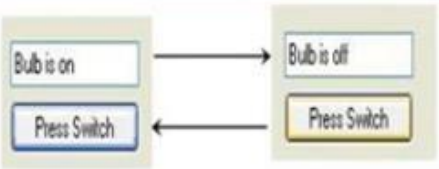
- They fall in to behavioral categories.
- State pattern allows an object to change its behavior depending on the current values of the object.



```
public class clsState
{
    enum BulbState
    {
        On = 1,
        Off = 2
    }
    private BulbState currentBulbState = BulbState.Off;
    public string getStatus()...

    public void PressSwitch()
    {
        if (currentBulbState == BulbState.Off)
        {
            currentBulbState = BulbState.On;
        }
        else if (currentBulbState == BulbState.On)
        {
            currentBulbState = BulbState.Off;
        }
    }
}
```

State class

State for the bulb



Client code consuming the state class

```
private void btnClick_Click(object sender, EventArgs e)
{
    objState.PressSwitch();
    txtState.Text = objState.getStatus();
}
```

Depending on state the behaviour of the objects changes

Call the 'PressSwitch' method from the client and display the status of the bulb

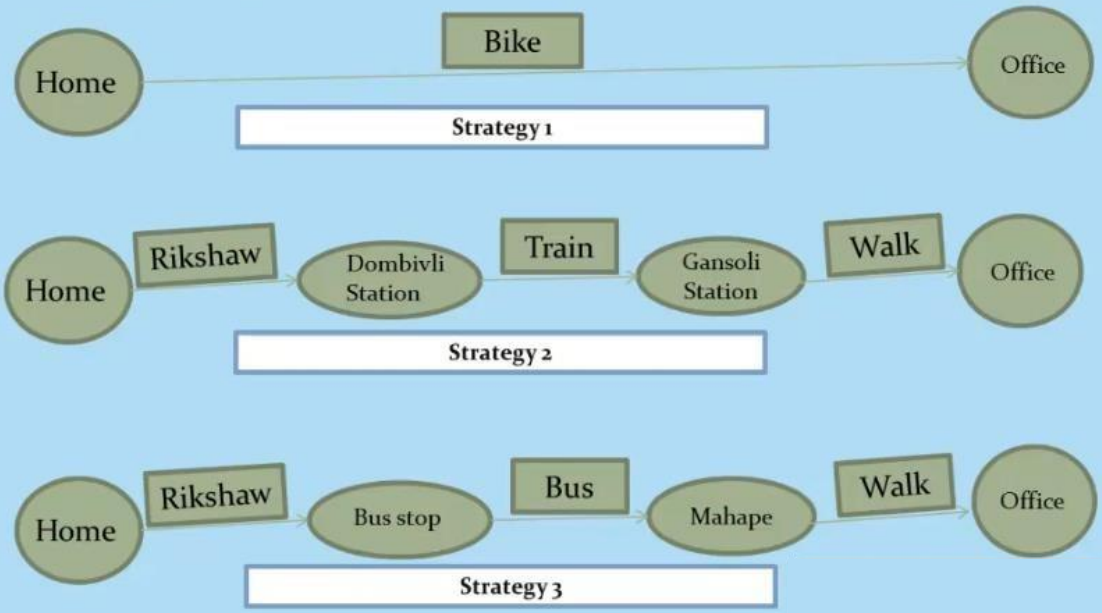
Strategy pattern

Agenda

- What is Strategy?
- What is Strategy Pattern?
- When to use?
- Practical demonstration and understand and how to use Strategy Pattern?

What is Strategy?

A strategy is a plan of action designed to achieve a specific goal



What is Strategy Pattern? When to use?

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

- Shopping Mall Example

1. Accept customer detail
2. Calculate bill amount
3. Apply discount based on day of week
   1. Monday – Low discount - 10%
   2. Thursday – High discount – 50%

What is Strategy Pattern? When to use?

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

- Shopping Mall E

1. Accept customer detail
2. Calculate bill amount
3. Apply discount based on day of week
   1. Monday – Low discount - 10%
   2. Thursday – High discount – 50%

- Customer Detail
- CalculateBill
- GetFinalBill

Shopping Mall will contain discount logic.
Showing Mall will contain reference to discount logic define

Showing Mall will contain reference to discount logic define

What is Strategy Pattern? When to use?

Open closed principle – software entities should be open for extension, but closed for modification.

New discount strategy may be applied in future

Non Reusable - Same discount strategy can be used in other shopping malls

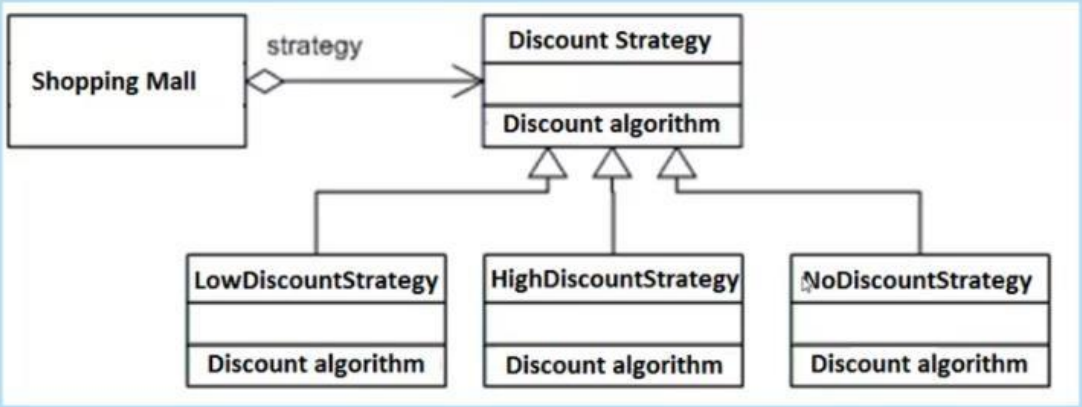
What is Strategy Pattern? When to use?

Open closed principle – software entities should be open for extension, but closed for modification.

New discount strategy may be applied in future

Non Reusable - Same discount strategy can be used in other shopping malls

- Program to an Interfaces not to an Implementaion



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace StrategyImplementaion
{
    public interface  IStrategy
    {
        int GetFinalBill(int BillAmount);
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace StrategyImplementaion
{
    public class LowDiscountStrategy:IStrategy
    {
        int IStrategy.GetFinalBill(int BillAmount)
        {
            return (int)(BillAmount-(BillAmount*0.1));
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace StrategyImplementaion
{
    public class HighDiscountStrategy:IStrategy
    {
        int IStrategy.GetFinalBill(int BillAmount)
        {
            return (int)(BillAmount-(BillAmount*0.5));
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace StrategyImplementaion
{
    public class NoDiscountStrategy:IStrategy
    {
        int IStrategy.GetFinalBill(int BillAmount)
        {
            return BillAmount;
        }
    }
}
```

```
using System.Text;

namespace StrategyImplementaion
{
    public class ShoppingMall
    {
        public string CustomerName { get; set; }
        public int BillAmount { get; set; }



        public IStrategy CurrentStrategy;
        public ShoppingMall(IStrategy NewStrategy)
        {
            CurrentStrategy = NewStrategy;
        }



        public int GetFinalBill()
        {
            return CurrentStrategy.GetFinalBill(this.BillAmount);
        }
    }
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
           //Today is Monday
            ShoppingMall objShoppingMall = new ShoppingMall(new LowDiscountStrategy());
            objShoppingMall.CustomerName = "Monday Customer";
            objShoppingMall.BillAmount = 1000;
            Console.WriteLine("Final Bill "+objShoppingMall.GetFinalBill());//10% discount

            //Today is Thursday
            ShoppingMall objShoppingMall2 = new ShoppingMall(new HighDiscountStrategy ());
            objShoppingMall2.CustomerName = "Thursday Customer";
            objShoppingMall2.BillAmount = 1000;
            Console.WriteLine("Final Bill " + objShoppingMall2.GetFinalBill());//

            //Today  is Sunday
            objShoppingMall2.CurrentStrategy = new NoDiscountStrategy();

            Console.WriteLine("Final Bill " + objShoppingMall2.GetFinalBill());

        }
    }
```

```
        static void Main(string[] args)
        {
            //Today is Monday
            ShoppingMall objShoppingMall = new ShoppingMall(new LowDiscountStrategy());
            objShoppingMall.CustomerName = "Monday Customer";
            objShoppingMall.BillAmount = 1000;
            Console.WriteLine("Final Bill "+objShoppingMal

            //Today is Thursday
            ShoppingMall objShoppingMall2 = new ShoppingMa
            objShoppingMall2.CustomerName = "Thursday Cust
            objShoppingMall2.BillAmount = 1000;
            Console.WriteLine("Final Bill " + objShoppingM

            //Today  is Sunday
            objShoppingMall2.CurrentStrategy = new NoDisco

            Console.WriteLine("Final Bill " + objShoppingM

        }
    }
}
```

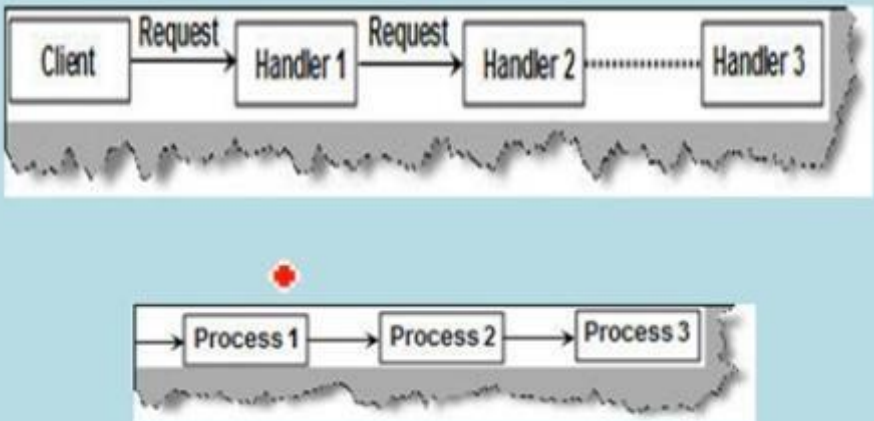
```
C:\Windows\system32\cmd.exe
Final Bill 900
Final Bill 500
Final Bill 1000
Press any key to continue . . .
```

```
        static void Main(string[] args)
        {
            ShoppingMall objShoppingMallGeneric = new ShoppingMall(null);
            objShoppingMallGeneric.CustomerName = "New Customer";
            objShoppingMallGeneric.BillAmount = 1000;

            switch (DateTime.Now.DayOfWeek)
            {

                case DayOfWeek.Monday:
                    objShoppingMallGeneric.CurrentStrategy = new LowDiscountStrategy();
                    break;
                case DayOfWeek.Thursday:
                    objShoppingMallGeneric.CurrentStrategy = new HighDiscountStrategy();
                    break;
                default:
                    objShoppingMallGeneric.CurrentStrategy = new NoDiscountStrategy();
                    break;
            }

            Console.WriteLine("Final Bill " + objShoppingMallGeneric.GetFinalBill());

        }
    }
```

CHAIN OF RESPONSIBILITY

Chain of responsibility (COR) Pattern

•Chain of responsibility is used when we have series of processing which will be handled by a series of handler logic




```
public class clsProcess1 : IProcess
{
    public override void runProcess()
    {
        Console.WriteLine("Run process 1");
        if (objProcess != null)
        {
            objProcess.runProcess();
        }
    }
}
```

Process 1

This defines the next process

```
lic abstract class IProcess

protected IProcess objProcess;
public void setProcess(IProcess _process)
{
    objProcess = _process;
}
public abstract void runProcess();
```

```
public class clsProcess2 : IProcess
{
    public override void runProcess()
    {
        Console.WriteLine("Run process 2");
        if (objProcess != null)
        {
            objProcess.runProcess();
        }
    }
```

Process 2

```
blic class clsProcess3 : IProcess

  public override void runProcess()
  {
      Console.WriteLine("Run process 3");
      if (objProcess != null)
      {
          objProcess.runProcess();
      }
  }
```

Process 3

These classes define their own implementation

```
clsProcess1 objProcess1 = new clsProcess1();
clsProcess2 objProcess2 = new clsProcess2();
clsProcess3 objProcess3 = new clsProcess3();

objProcess1.setProcess(objProcess2);

objProcess2.setProcess(objProcess3);

objProcess1.runProcess();
Console.ReadLine();
```

Create all objects

Set the process link list

Run the process