

# Design Pattern FAQ - Part 1 (Training)



**Shivprasad koirala**

9 Jun 2021 CPOL

Rate me: ★★★★★ 4.46/5 (179 votes)

Design Pattern FAQ - Interview Questions (Quick Reference)

This article is Part 1 of a series of 4 articles on Frequency Asked Questions about Design Patterns.

Updated with the explanation of Singleton pattern.

## Design Pattern FAQs

- [What are design patterns?](#)
- [Can you explain factory pattern?](#)
- [Can you explain abstract factory pattern?](#)
- [Can you explain builder pattern?](#)
- [Can you explain prototype pattern?](#)
- [Can you explain shallow copy and deep copy in prototype patterns?](#)
- [Can you explain singleton pattern?](#)
- [Can you explain command patterns?](#)
- [Design Pattern with a Project](#)

## Introduction

Here's a small quick FAQ on design patterns in Q and A format. In this section, we will cover factory, abstract factory, builder, prototype, shallow and deep prototype, singleton and command patterns.

You can read my other design pattern FAQ sections of the same in the below links:

- Part 2 Design Pattern FAQs -- [Interpreter pattern](#), [iterator pattern](#), [mediator pattern](#), [memento pattern](#) and [observer pattern](#)
- Part 3 Design Pattern FAQs -- [State pattern](#), [strategy pattern](#), [visitor pattern](#), [adapter pattern](#) and [fly weight pattern](#)
- Part 4 Design Pattern FAQs -- [Bridge pattern](#), [composite pattern](#), [decorator pattern](#), [Façade pattern](#), [chain of responsibility \(COR\)](#), [proxy pattern](#) and [template pattern](#)

## What Are Design Patterns?

Design patterns are documented, tried and tested solutions for recurring problems in a given context. So basically, you have a problem context and the proposed solution for the same. Design patterns existed in some or other form right from the inception stage of software development.

Let's say if you want to implement a sorting algorithm, the first thing that comes to mind is bubble sort. So the problem is sorting and solution is bubble sort. Same holds true for design patterns.

## Which are the Three Main Categories of Design Patterns?

There are three basic classifications of patterns - Creational, Structural, and Behavioral patterns.

### Creational Patterns

- **Abstract Factory:** Creates an instance of several families of classes
- **Builder:** Separates object construction from its representation
- **Factory Method:** Creates an instance of several derived classes
- **Prototype:** A fully initialized instance to be copied or cloned
- **Singleton:** A class in which only a single instance can exist

**Note:** The best way to remember Creational pattern is by remembering ABFPS (Abraham Became First President of States).

### Structural Patterns

- **Adapter:** Match interfaces of different classes
- **Bridge:** Separates an object's abstraction from its implementation
- **Composite:** A tree structure of simple and composite objects
- **Decorator:** Add responsibilities to objects dynamically
- **Facade:** A single class that represents an entire subsystem
- **Flyweight:** A fine-grained instance used for efficient sharing
- **Proxy:** An object representing another object
- **Mediator:** Defines simplified communication between classes
- **Memento:** Capture and restore an object's internal state
- **Interpreter:** A way to include language elements in a program
- **Iterator:** Sequentially access the elements of a collection
- **Chain of Resp:** A way of passing a request between a chain of objects
- **Command:** Encapsulate a command request as an object
- **State:** Alter an object's behavior when its state changes
- **Strategy:** Encapsulates an algorithm inside a class
- **Observer:** A way of notifying change to a number of classes
- **Template Method:** Defer the exact steps of an algorithm to a subclass
- **Visitor:** Defines a new operation to a class without change

## Can You Explain Factory Pattern?

Factory pattern is one of the types of creational patterns. You can make out from the name factory itself that it's meant to construct and create something. In software architecture world factory pattern is meant to centralize creation of objects. Below is a code snippet of a client which has different types of invoices. These invoices are created depending on the invoice type specified by the client. There are two issues with the code below:

- First, we have lots of 'new' keywords scattered in the client. In other ways, the client is loaded with lot of object creational activities which can make the client logic very complicated.
- The second issue is that the client needs to be aware of all types of invoices. So if we are adding one more invoice class type called as 'InvoiceWithFooter', we need to reference the new class in the client and recompile the client also.

```

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

```

**Figure: Different types of invoice**

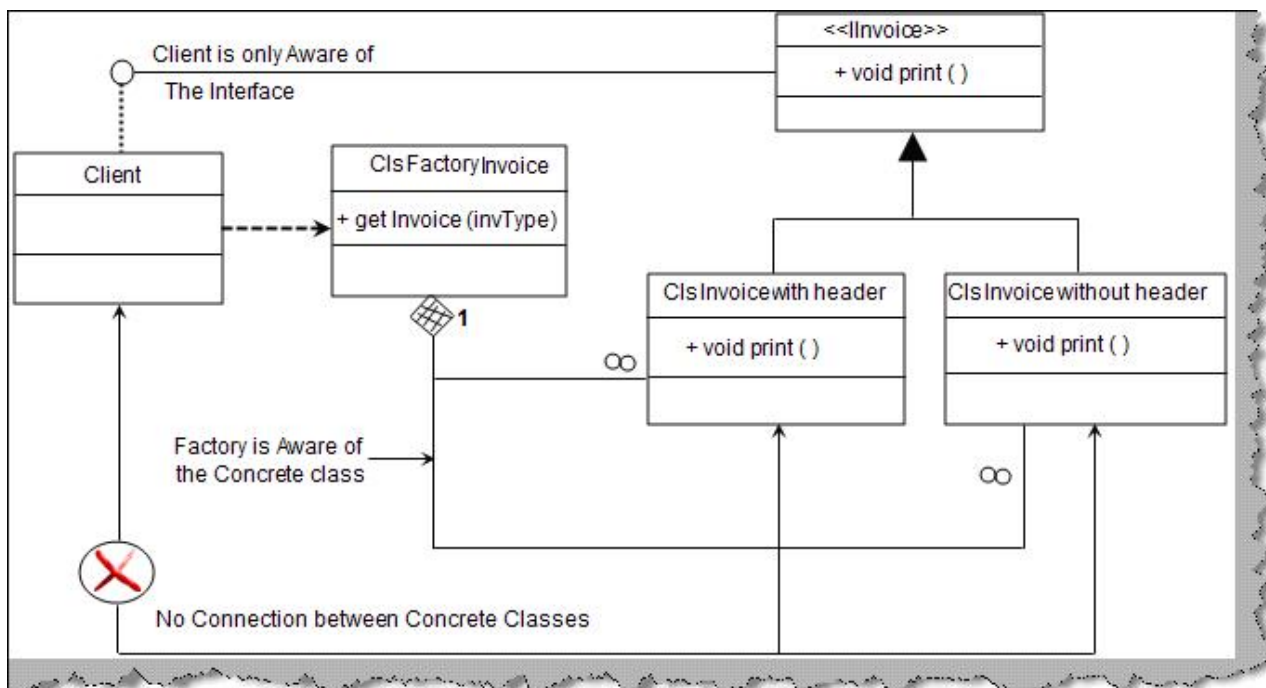
Taking these issues as our base, we will now look into how factory pattern can help us solve the same. The below figure 'Factory Pattern' shows two concrete classes 'ClsInvoiceWithHeader' and 'ClsInvoiceWithoutHeader'.

The **first issue** was that these classes are in direct contact with client which leads to lot of the 'new' keyword scattered in the client code. This is removed by introducing a new class 'ClsFactoryInvoice' which does all the creation of objects.

The **second issue** was that the client code is aware of both the concrete classes, i.e., 'ClsInvoiceWithHeader' and 'ClsInvoiceWithoutHeader'. This leads to recompiling of the client code when we add new invoice types. For instance, if we add 'ClsInvoiceWithFooter', the client code needs to be changed and recompiled accordingly. To remove this issue, we have introduced a common interface 'IInvoice'. Both the concrete classes, 'ClsInvoiceWithHeader' and 'ClsInvoiceWithoutHeader' inherit and implement the 'IInvoice' interface.

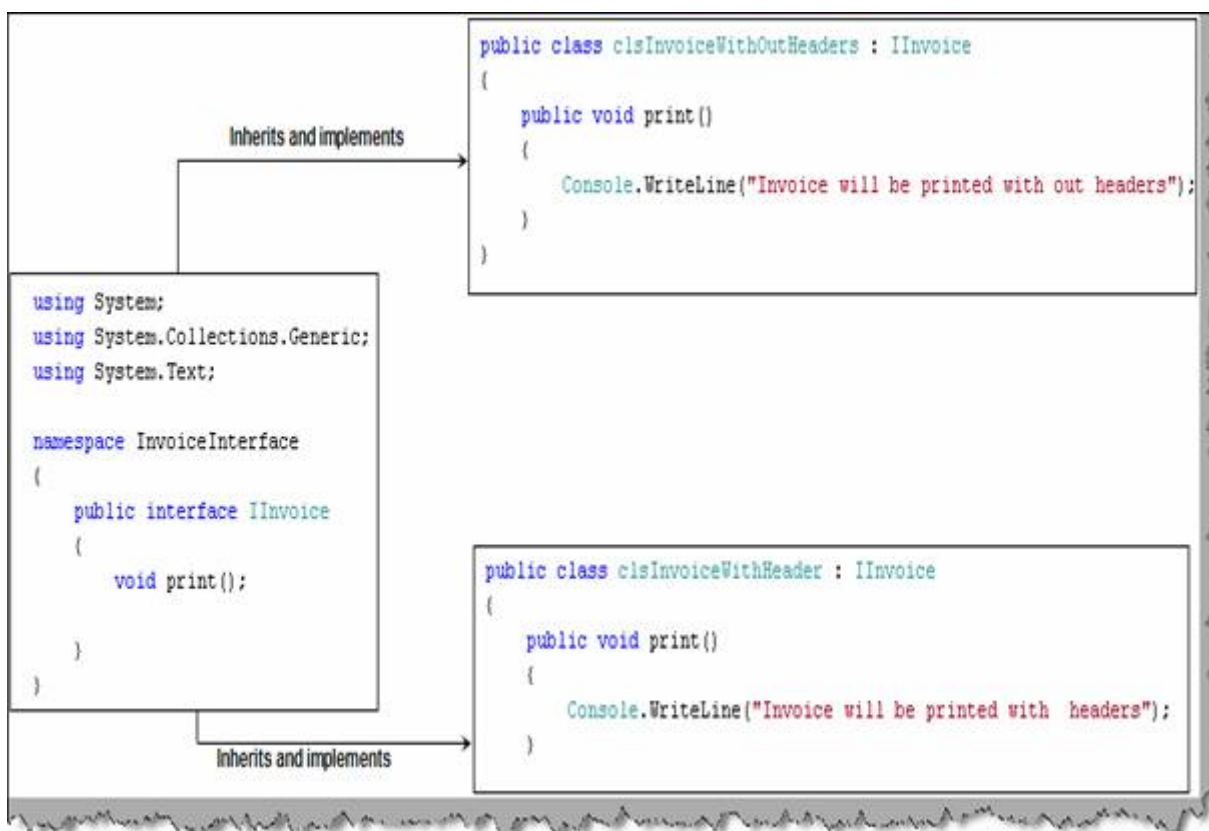
The client references only the 'IInvoice' interface which results in zero connection between client and the concrete classes ('ClsInvoiceWithHeader' and 'ClsInvoiceWithoutHeader'). So now, if we add new concrete invoice class, we do not need to change anything on the client side.

In one line, the creation of objects is taken care by 'ClsFactoryInvoice' and the client disconnection from the concrete classes is taken care by 'IInvoice' interface.



**Figure: Factory pattern**

Below are the code snippets of how actually factory pattern can be implemented in C#. In order to avoid recompiling the client, we have introduced the invoice interface 'IInvoice'. Both the concrete classes 'ClsInvoiceWithOutHeaders' and 'ClsInvoiceWithHeader' inherit and implement the 'IInvoice' interface.



**Figure: Interface and concrete classes**

We have also introduced an extra class 'ClsFactoryInvoice' with a function 'getInvoice()' which will generate objects of both the invoices depending on 'intInvoiceType' value. In short, we have centralized the logic of object creation in the 'ClsFactoryInvoice'. The client calls the 'getInvoice'

function to generate the invoice classes. One of the most important points to be noted is that client only refers to 'IInvoice' type and the factory class 'ClsFactoryInvoice' also gives the same type of reference. This helps the client to be completely detached from the concrete classes, so now when we add new classes and invoice types, we do not need to recompile the client.

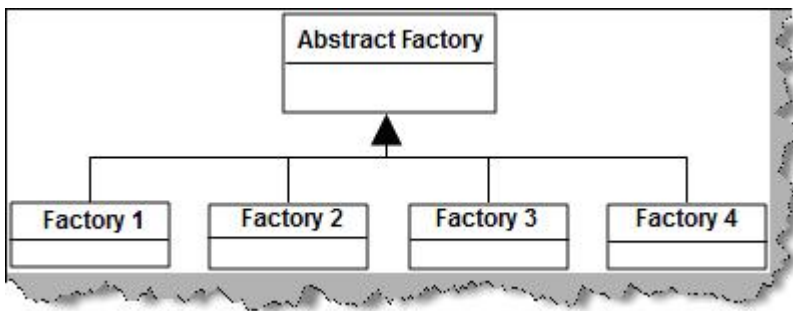


**Figure: Factory class which generates objects**

## Can You Explain Abstract Factory Pattern?

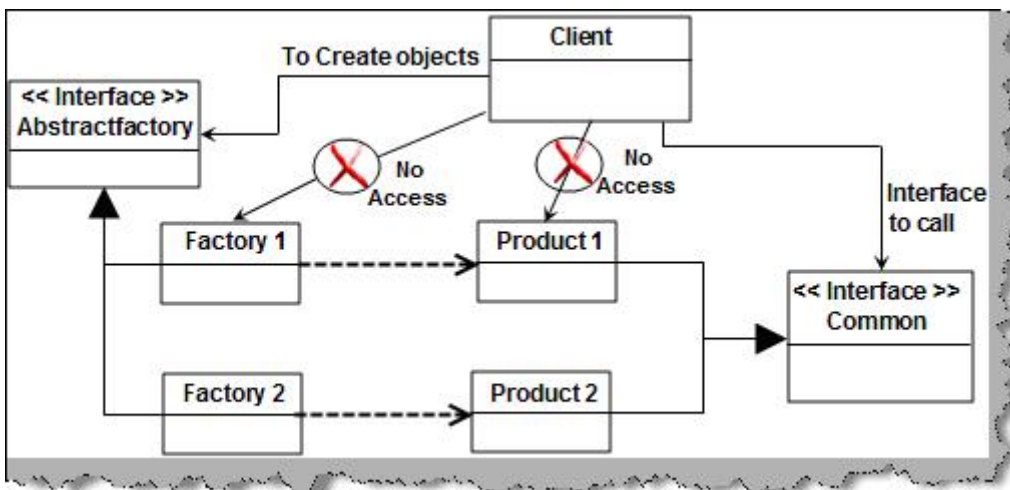
Abstract factory expands on the basic factory pattern. Abstract factory helps us to unite similar factory pattern classes into one unified interface. So basically, all the common factory patterns now inherit from a common abstract factory class which unifies them in a common class. All other things related to factory pattern remain same as discussed in the previous question.

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



**Figure: Abstract factory unifies related factory patterns**

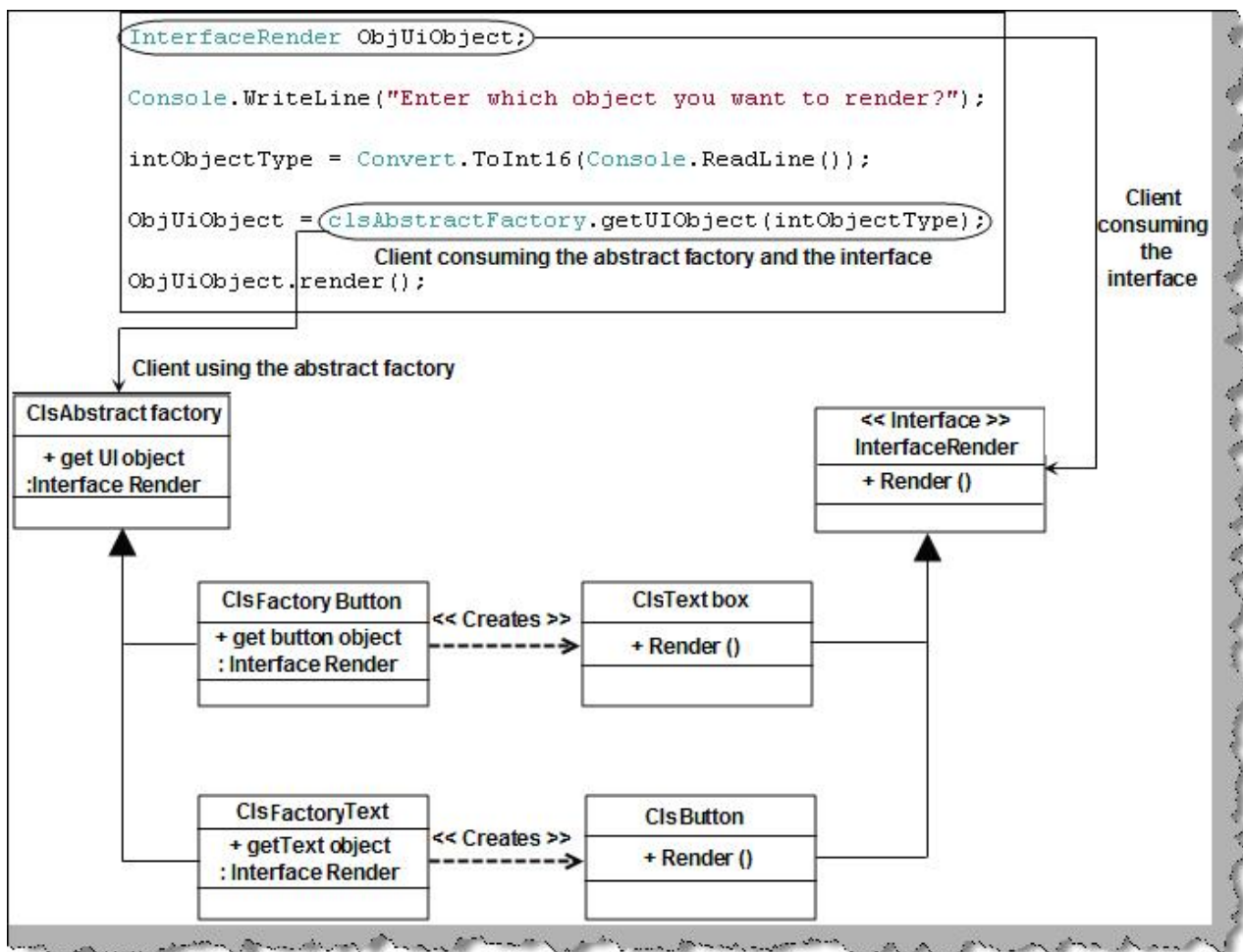
Now that we know the basics, let's try to understand the details of how abstract factory patterns are actually implemented. As said previously, we have the factory pattern classes (**factory1** and **factory2**) tied up to a common **abstract** factory (**AbstractFactory** Interface) via inheritance. **Factory** classes stand on top of concrete classes which are again derived from common interface. For instance, in figure 'Implementation of abstract factory', both the concrete classes, '**product1**' and '**product2**' inherit from one interface, i.e., '**common**'. The client who wants to use the concrete class will only interact with the abstract factory and the common interface from which the concrete classes inherit.



**Figure: Implementation of abstract factory**

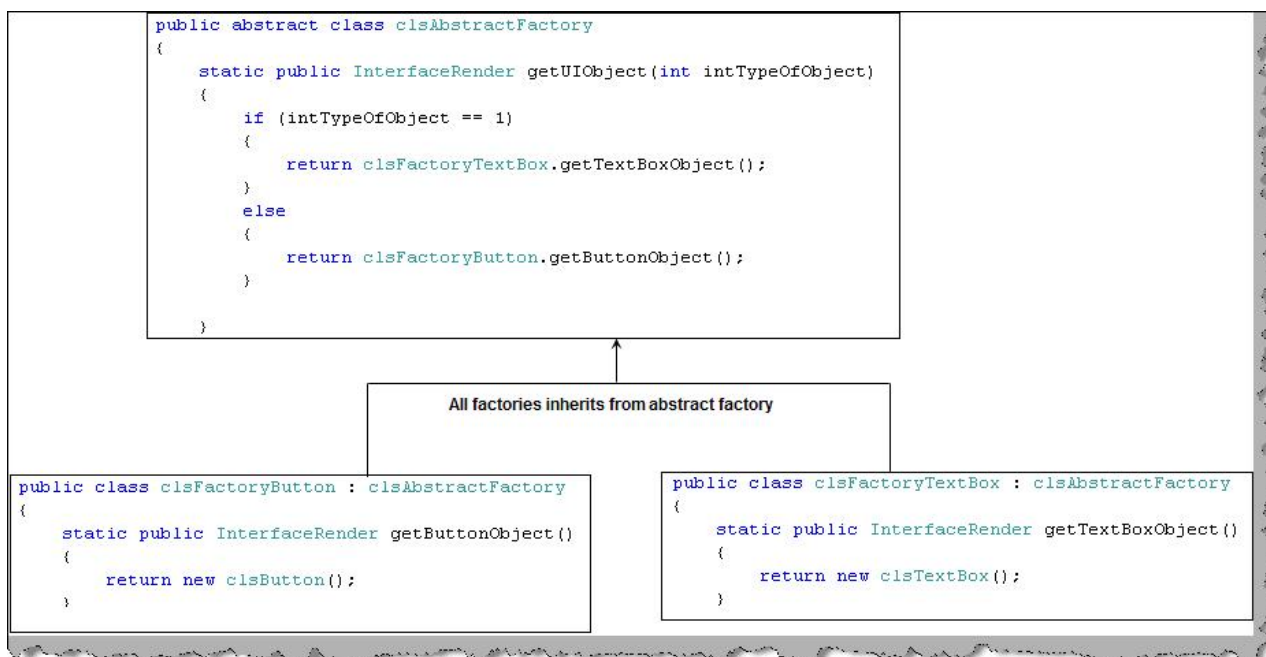
Now let's have a look at how we can practically implement abstract factory in actual code. We have a scenario where we have UI creational activities for textboxes and buttons through their own centralized factory classes, '**ClsFactoryButton**' and '**ClsFactoryText**'. Both these classes inherit from common interface '**InterfaceRender**'. Both the factories '**ClsFactoryButton**' and '**ClsFactoryText**' inherits from the common factory '**ClsAbstractFactory**'. Figure 'Example for **AbstractFactory**' shows how these classes are arranged and the client code for the same. One of the important points to be noted about the client code is that it does not interact with the concrete classes. For object creation, it uses the abstract factory (**ClsAbstractFactory**) and for calling the concrete class implementation, it calls the methods via the interface '**InterfaceRender**'. So the '**ClsAbstractFactory**' class provides a common interface for both factories '**ClsFactoryButton**' and '**ClsFactoryText**'.





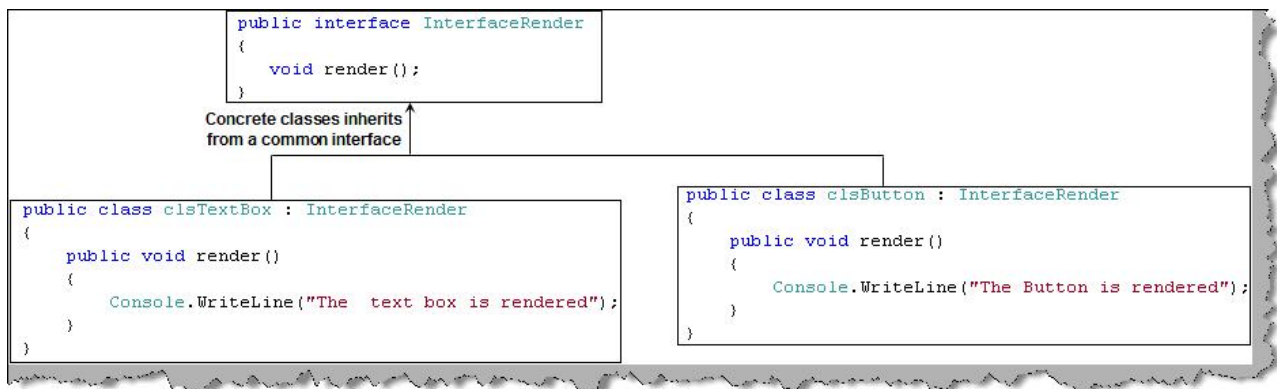
**Figure: Example for abstract factory**

We will just run through the sample code for abstract factory. The below code snippet 'Abstract factory and factory code snippet' shows how the factory pattern classes inherit from abstract factory.



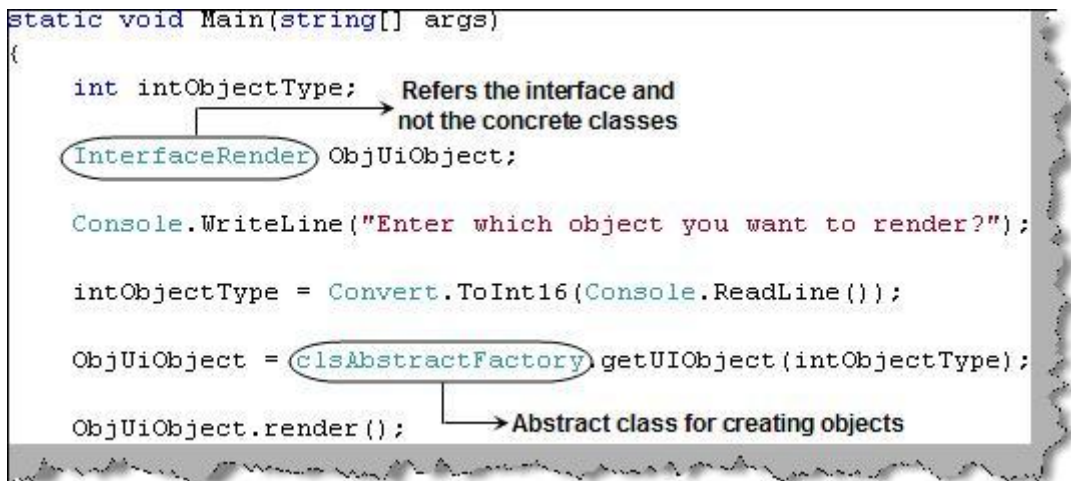
**Figure: Abstract factory and factory code snippet**

Figure 'Common Interface for concrete classes' how the concrete classes inherits from a common interface 'InterfaceRender' which enforces the method 'render' in all the concrete classes.



**Figure: Common interface for concrete classes**

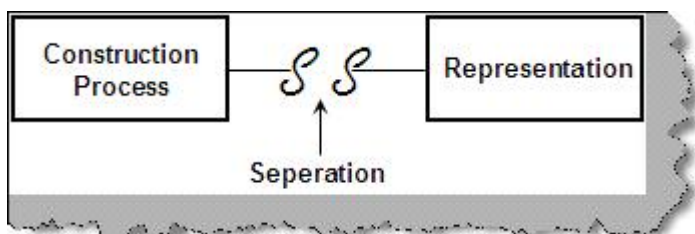
The final thing is the client code which uses the interface 'InterfaceRender' and abstract factory 'ClsAbstractFactory' to call and create the objects. One of the important points about the code is that it is completely isolated from the concrete classes. Due to this, any changes in concrete classes like adding and removing concrete classes does not need client level changes.



**Figure: Client, interface and abstract factory**

## Can You Explain Builder Pattern?

Builder falls under the type of creational pattern category. 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. The main objective is to separate the construction of objects and their representations. If we are able to separate the construction and representation, we can then get many representations from the same construction.

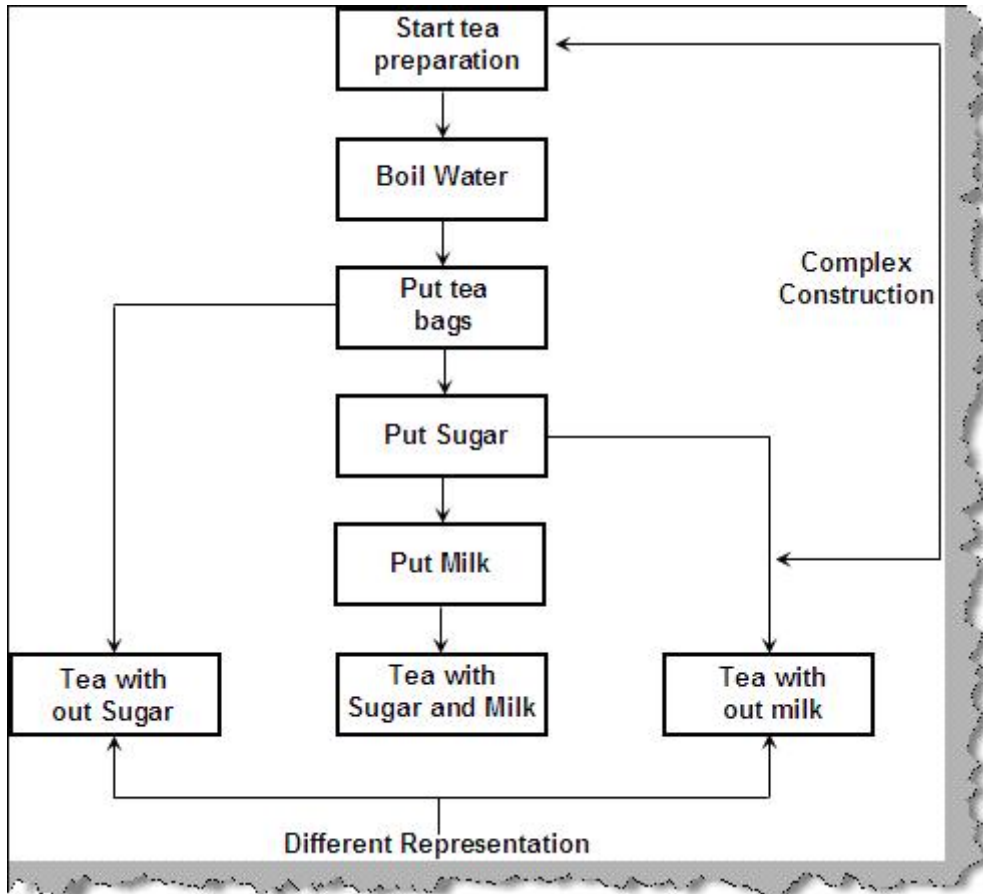




**Figure: Builder concept**

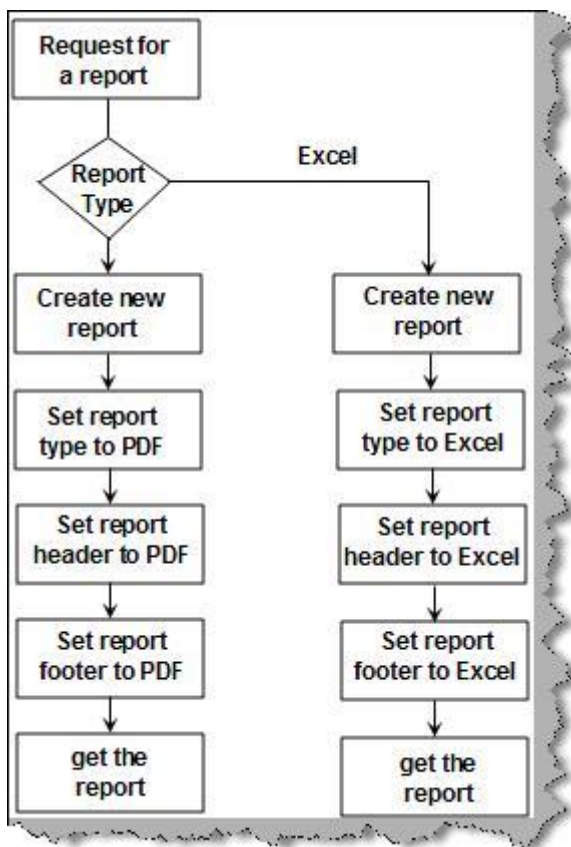
To understand what we mean by construction and representation, let's take the example of the below 'Tea preparation' sequence.

You can see from the figure 'Tea preparation' from the same preparation steps, we can get three representations of teas (i.e., tea without sugar, tea with sugar / milk and tea without milk).



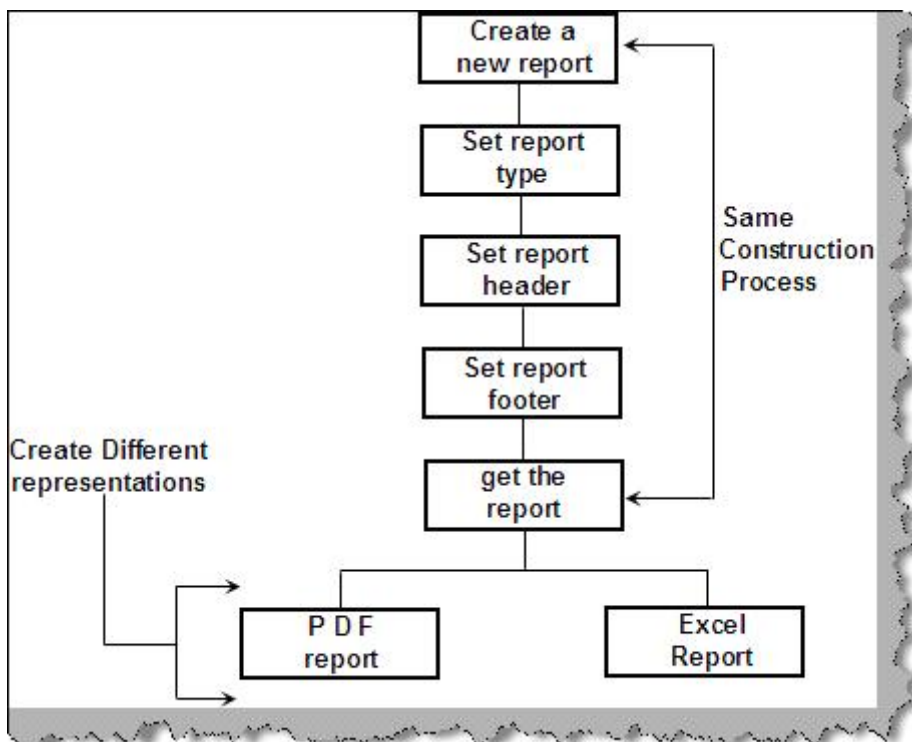
**Figure: Tea preparation**

Now let's take a real time example in the software world to see how builder can separate the complex creation and its representation. Consider we have an application where we need the same report to be displayed in either 'PDF' or 'EXCEL' format. Figure 'Request a report' shows the series of steps to achieve the same. Depending on report type, a new report is created, report type is set, headers and footers of the report are set and finally, we get the report for display.



**Figure: Request a report**

Now let's take a different view of the problem as shown in figure 'Different View'. The same flow defined in 'Request a report' is now analyzed in representations and common construction. The construction process is the same for both the types of reports but they result in different representations.

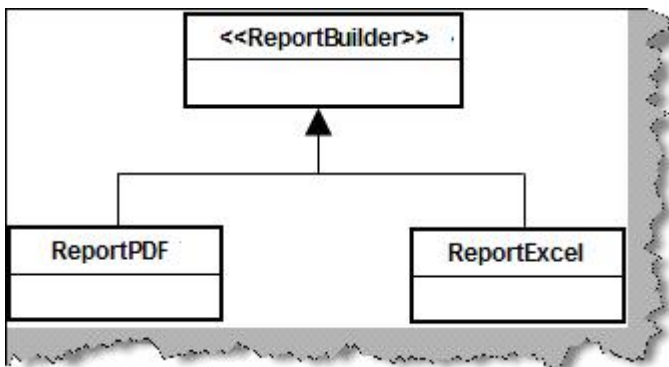


**Figure: Different View**

We will take the same report problem and try to solve the same using builder patterns. There are three main parts when you want to implement builder patterns:

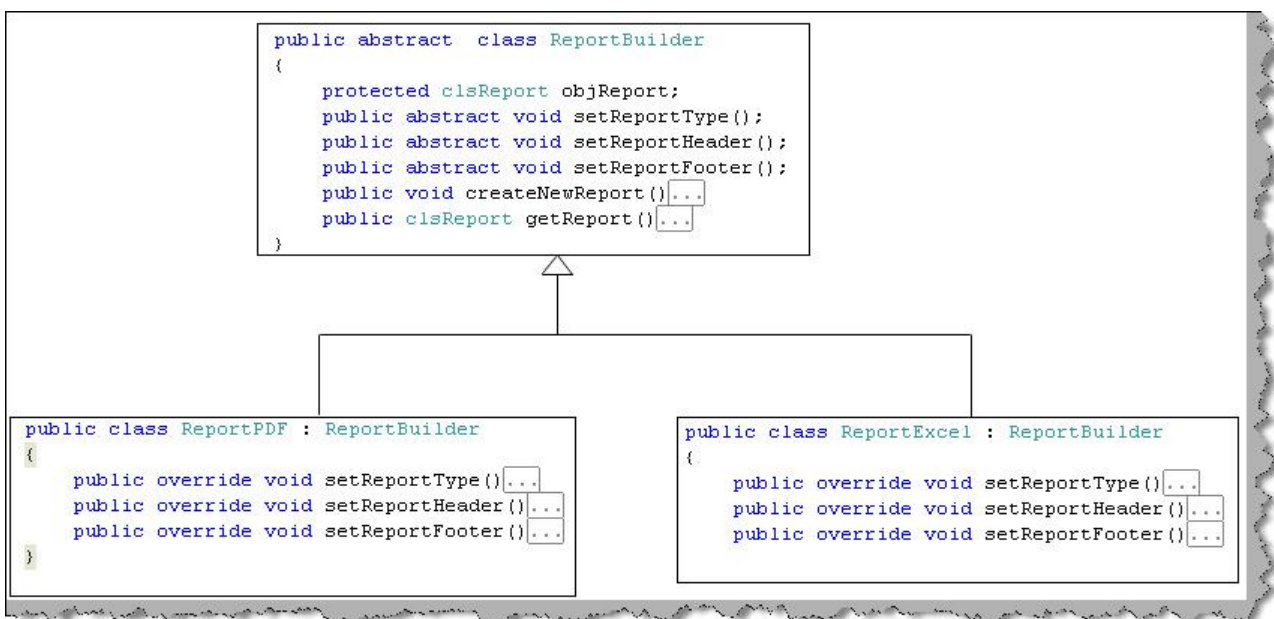
- **Builder**: **Builder** is responsible for defining the construction process for individual parts. Builder has those individual processes to initialize and configure the product.
- **Director**: **Director** takes those individual processes from the builder and defines the sequence to build the product.
- **Product**: **Product** is the final object which is produced from the builder and director coordination.

First, let's have a look at the builder class hierarchy. We have an **abstract** class called as '**ReportBuilder**' from which custom builders like '**ReportPDF**' builder and '**ReportEXCEL**' builder will be built.



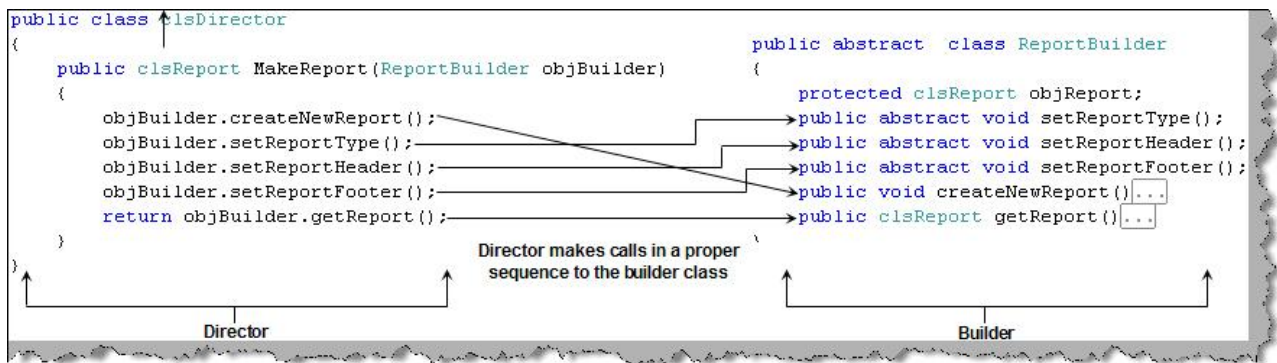
**Figure: Builder class hierarchy**

Figure 'Builder classes in actual code' shows the methods of the classes. To generate report, we need to first Create a new report, set the report type (to EXCEL or PDF), set report headers, set the report footers and finally get the report. We have defined two custom builders one for 'PDF' (**ReportPDF**) and other for 'EXCEL' (**ReportExcel**). These two custom builders define their own process according to the report type.



**Figure: Builder classes in actual code**

Now let's understand how director will work. Class '**clsDirector**' takes the builder and calls the individual method process in a sequential manner. So director is like a driver who takes all the individual processes and calls them in sequential manner to generate the final product, which is the report in this case. Figure 'Director in action' shows how the method '**MakeReport**' calls the individual process to generate the report product by PDF or EXCEL.



**Figure: Director in action**

The third component in the builder is the product which is nothing but the report class in this case.

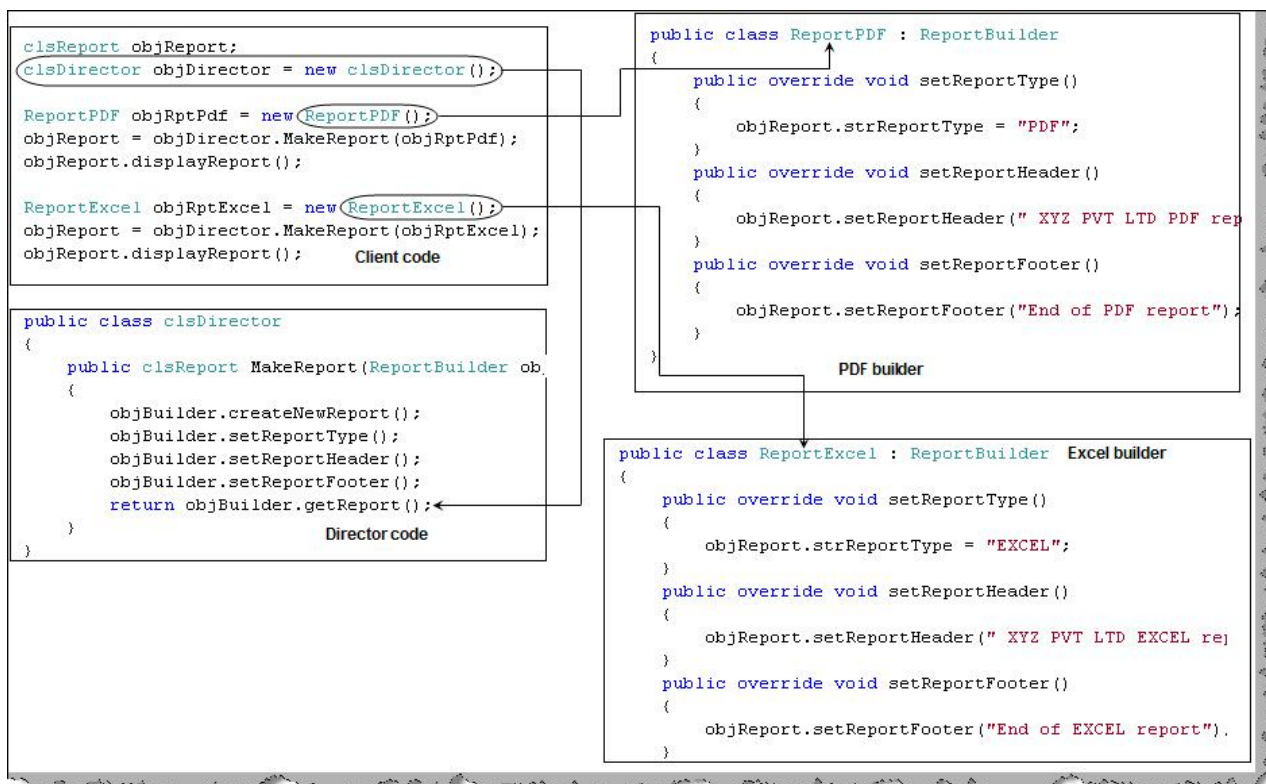
```

public class clsReport
{
    public string strReportType;
    private ArrayList objHeader = new ArrayList();
    private ArrayList objFooter = new ArrayList();
    public void setReportFooter(string strData)...
    public void setReportHeader(string strData)...
    public void displayReport()...
}

```

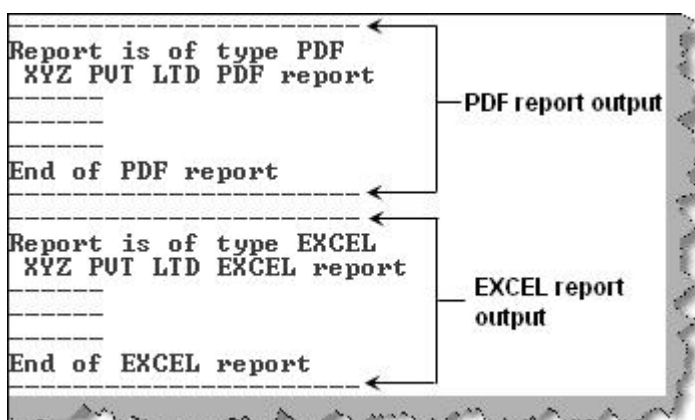
**Figure: The report class**

Now let's take a top view of the builder project. Figure 'Client, builder, director and product' shows how they work to achieve the builder pattern. Client creates the object of the director class and passes the appropriate builder to initialize the product. Depending on the builder, the product is initialized/created and finally sent to the client.



**Figure: Client, builder, director and product**

The output is something like this. We can see two report types displayed with their headers according to the builder.



**Figure: Final output of builder**

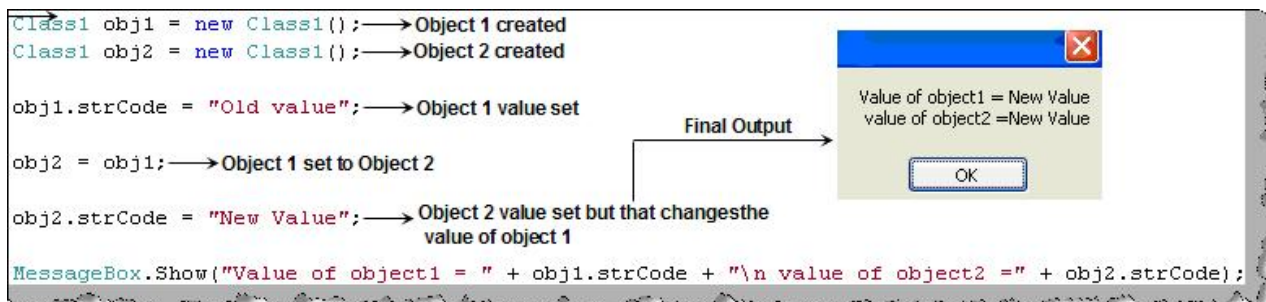
## Can You Explain Prototype Pattern?

Prototype pattern falls in the section of creational pattern. 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 value. If you are thinking by just setting objects, we can get a clone, then you have mistaken it. By setting one object to other object, we set the reference of object **BYREF**. So changing the new object also changed the original object. To understand the **BYREF** fundamental more clearly, consider the figure 'BYREF' below. Following is the sequence of the below code:

- In the first step, we have created the first object, i.e., **obj1** from **class1**.
- In the second step, we have created the second object, i.e., **obj2** from **class1**.



- In the third step, we set the values of the old object, i.e., **obj1** to 'old value'.
- In the fourth step, we set the **obj1** to **obj2**.
- In the fifth step, we change the **obj2** value.
- Now, we display both the values and we have found that both the objects have the new value.

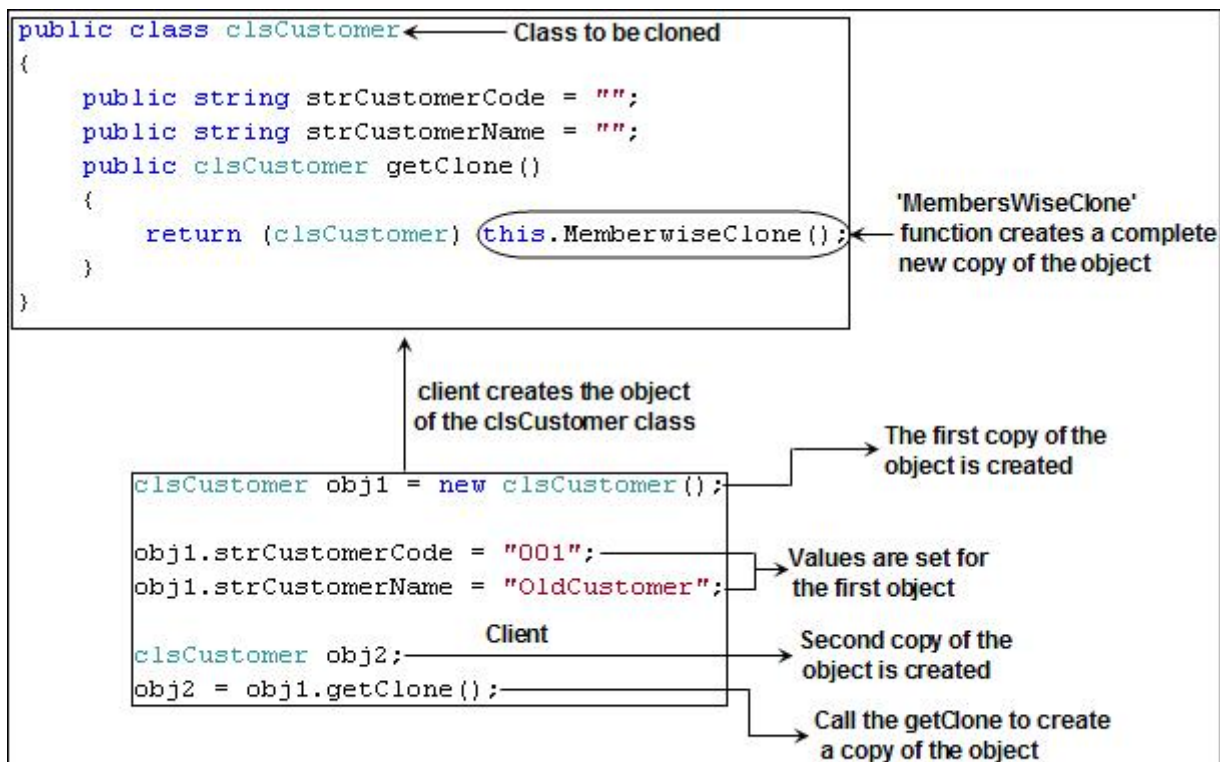


**Figure: BYREF**

The conclusion of the above example is that objects when set to other objects are set **BYREF**. So changing new object values also changes the old object value.

There are many instances when we want the new copy object changes should not affect the old object. The answer to this is prototype patterns.

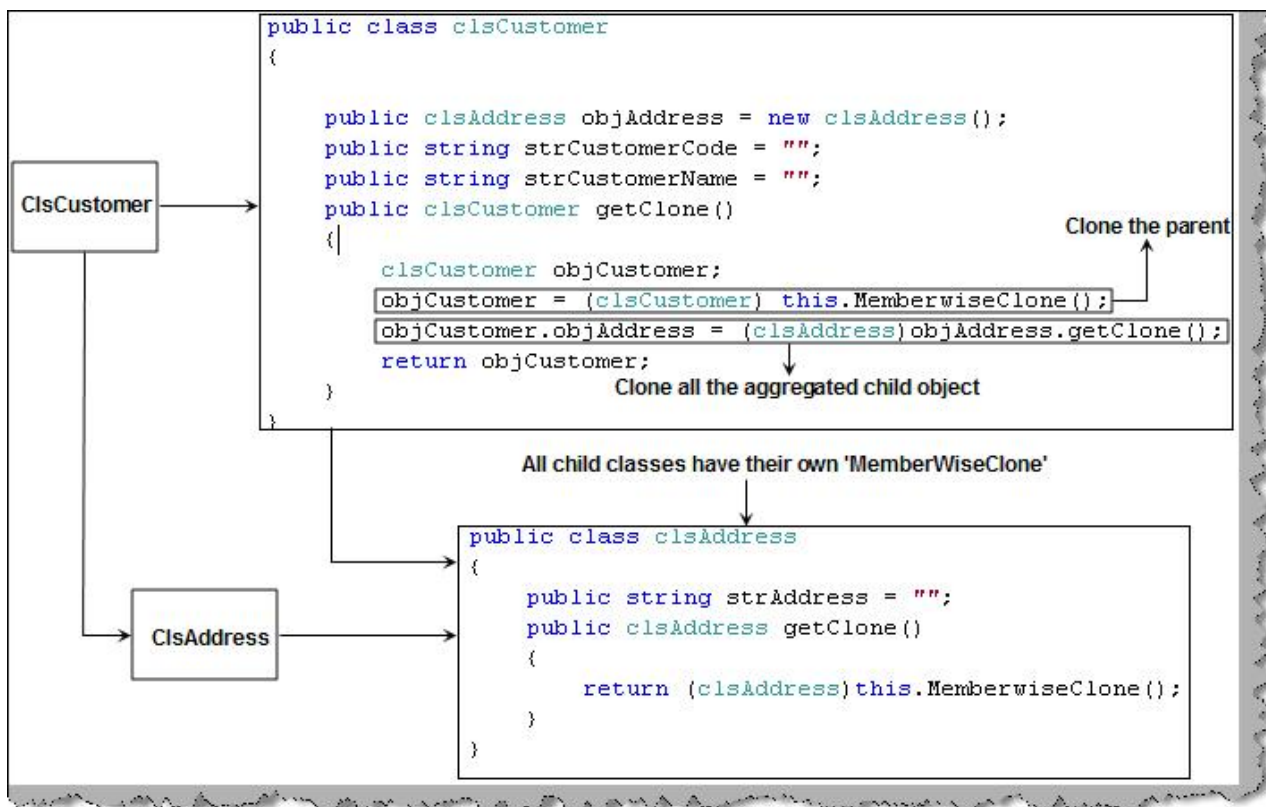
Let's look at how we can achieve the same using C#. In the below figure 'Prototype in action', we have the customer class '**ClsCustomer**' which needs to be cloned. This can be achieved in C# by using the '**MemberWiseClone**' method. In JAVA, we have the '**Clone**' method to achieve the same. In the same code, we have also shown the client code. We have created two objects of the customer class '**obj1**' and '**obj2**'. Any changes to '**obj2**' will not affect '**obj1**' as it's a complete cloned copy.



**Figure: Prototype in action**

## Can You Explain Shallow Copy and Deep Copy in Prototype Patterns?

There are two types of cloning for prototype patterns. One is the shallow cloning which you have just read in the first question. In shallow copy, only that object is cloned, any objects containing in that object is not cloned. For instance, consider the figure 'Deep cloning in action', we have a **customer** class and we have an **address** class aggregated inside the **customer** class. 'MemberWiseClone' will only clone the **customer** class 'ClsCustomer' but not the 'ClsAddress' class. So we added the 'MemberWiseClone' function in the **address** class also. Now when we call the 'getClone' function, we call the parent cloning function and also the child cloning function, which leads to cloning of the complete object. When the parent objects are cloned with their containing objects, it's called as deep cloning and when only the parent is cloned, it's termed as shallow cloning.



**Figure: Deep cloning in action**

## Can You Explain Singleton Pattern?

Punch :- Create a single instance of object and provides access to this single instance via a central point.

There are situations in project where we want only one instance of the object to be created and shared between the clients. For instance, let's say we have the below two classes, **currency** and **country**.

These classes load master data which will be referred again and again in project. We would like to share a single instance of the class to get performance benefit by not hitting the database again and again.

C#



```
public class Currency
{
    List<string> oCurrencies = new List<string>();
    public Currency()
    {
        oCurrencies.Add("INR");
        oCurrencies.Add("USD");
        oCurrencies.Add("NEP");
        oCurrencies.Add("GBP");
    }
    public IEnumerable<string> getCurrencies()
    {
        return (IEnumerable<string>)oCurrencies;
    }
}
```

C#



```
public class Country
{
    List<string> oCountries = new List<string>();
    public Country()
    {
        oCountries.Add("India");
        oCountries.Add("Nepal");
        oCountries.Add("USA");
        oCountries.Add("UK");
    }
    public IEnumerable<string> getCounties()
    {
        return (IEnumerable<string>) oCountries;
    }
}
```

Implementing singleton pattern is a four step process.

### Step 1: Create a sealed class with a private constructor

The **private** constructor is important so that clients cannot create object of the class directly. If you remember the punch, the main intention of this pattern is to create a single instance of the object which can be shared globally, so we do not want to give client the control to create instances directly.

C#



```
public sealed class GlobalSingleton
{
    private GlobalSingleton() { }

    .....
    .....
}
```

**Step 2: Create aggregated objects of the classes (for this example, it is currency and country) for which we want single instance.**

C#



```
public sealed class GlobalSingleton
{
    // object which needs to be shared globally
    public Currency Currencies = new Currency();
    public Country Countries = new Country();
}
```

**Step 3: Create a static read-only object of the class itself and expose the same via static property as shown below.**

C#



```
public sealed class GlobalSingleton
{
    ...
    ...
    // use static variable to create a single instance of the object
    static readonly GlobalSingleton INSTANCE = new GlobalSingleton();

    public static GlobalSingleton Instance
    {
        get
        {
            return INSTANCE;
        }
    }
}
```

**Step 4: You can now consume the singleton object using the below code from the client.**

C#



```
GlobalSingleton oSingle = GlobalSingleton.Instance;
Country o = oSingle.Country;
```

Below goes the complete code for singleton pattern which we discussed above in pieces.

C#



```
public sealed class GlobalSingleton
{
    // object which needs to be shared globally
    public Currency Currencies = new Currency();
    public Country Countries = new Country();

    // use static variable to create a single instance of the object
    static readonly GlobalSingleton INSTANCE = new GlobalSingleton();

    /// This is a private constructor, meaning no outsiders have access.
}
```

```
private GlobalSingleton()  
{ }  
  
public static GlobalSingleton Instance  
{  
    get  
    {  
        return INSTANCE;  
    }  
}
```

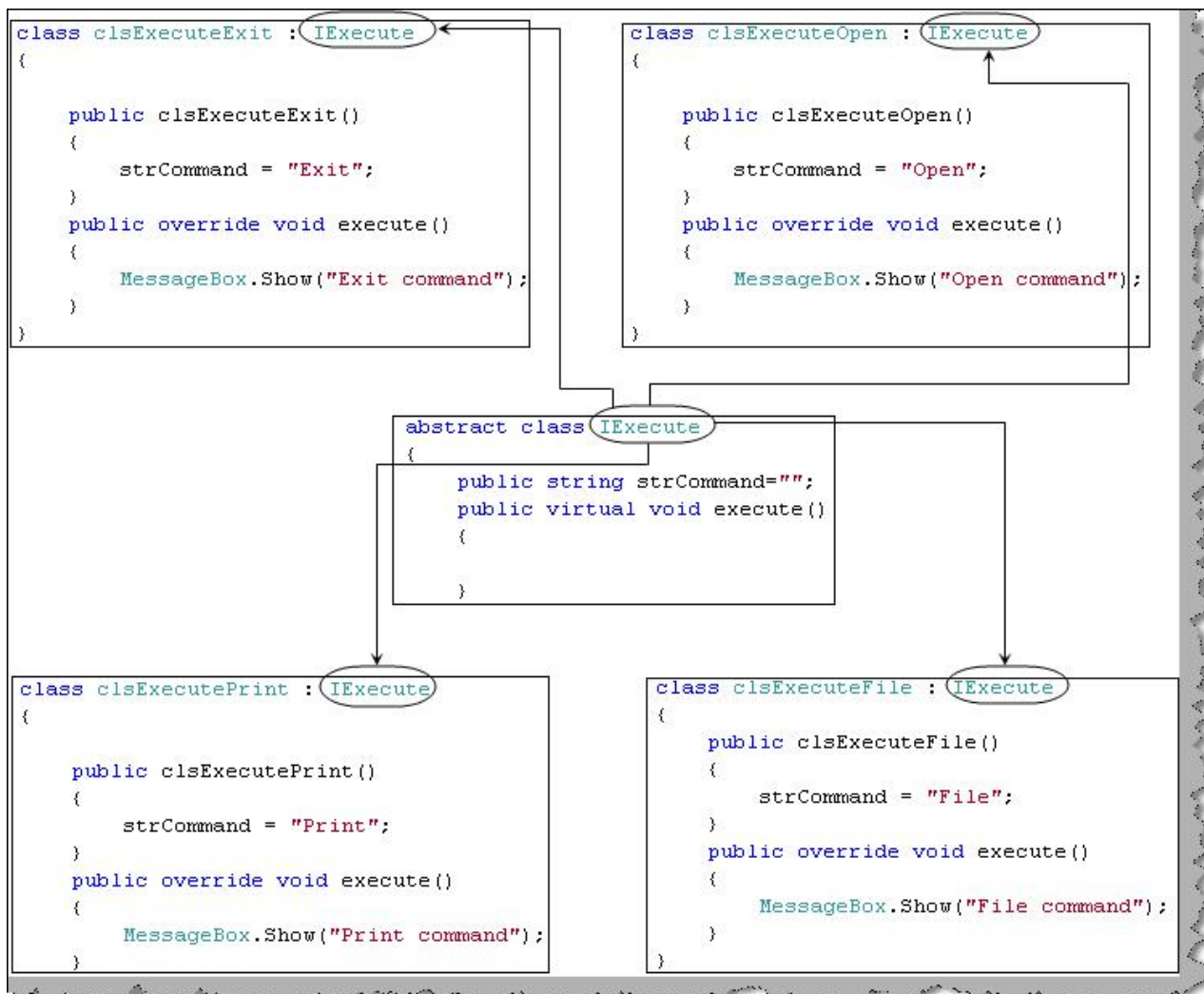
## Can You Explain Command Patterns?

Command pattern allows a request to exist as an object. Ok, let's understand what it means. Consider the figure 'Menu and Commands', we have different actions depending on which menu is clicked. So depending on which menu is clicked, we have passed a **string** which will have the action text in the action **string**. Depending on the action **string**, we will execute the action. The bad thing about the code is it has lot of '**IF**' conditions which makes the coding more cryptic.



Command pattern moves the above action in to objects. These objects when executed actually execute the command.

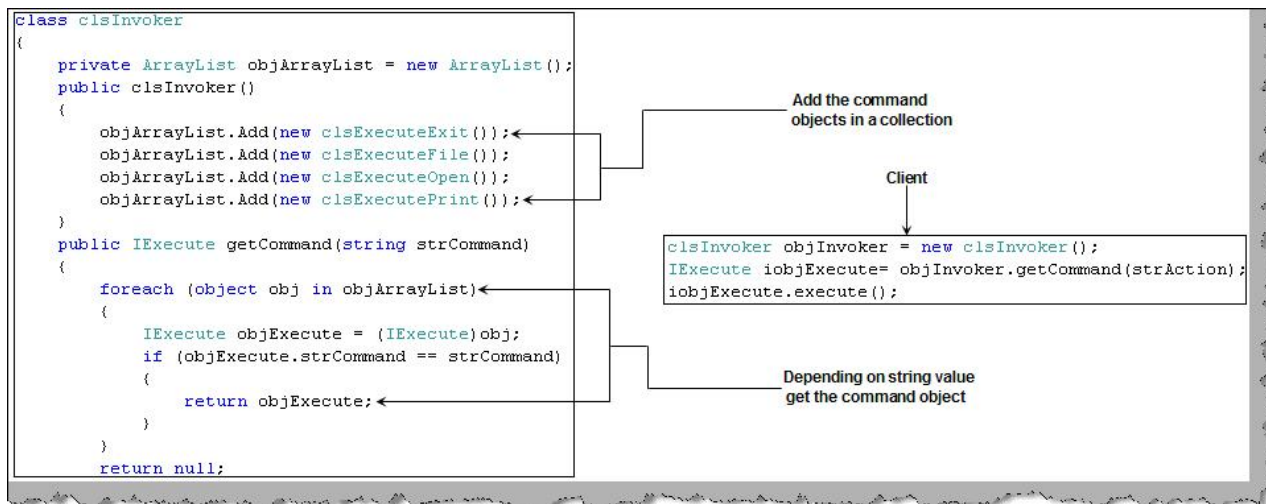
As said previously, every command is an object. We first prepare individual classes for every action, i.e., exit, open, file and print. All the above actions are wrapped in to classes like **Exit** action is wrapped in '**clsExecuteExit**', open action is wrapped in '**clsExecuteOpen**', print action is wrapped in '**clsExecutePrint**' and so on. All these classes are inherited from a common interface '**IExecute**'.



**Figure: Objects and Command**

Using all the action classes, we can now make the invoker. The main work of invoker is to map the action with the classes which have the action.

So we have added all the actions in one collection, i.e., the **arraylist**. We have exposed a method '**getCommand**' which takes a **string** and gives back the **abstract** object '**IExecute**'. The client code is now neat and clean. All the '**IF**' conditions are now moved to the '**clsInvoker**' class.



**Figure: Invoker and the clean client**

In case you are completely new to design patterns or you really do not want to read this complete article, do see our free [design pattern Training and interview questions / answers](#) videos.

# Design pattern FAQ Part 2 (Design pattern training series)



**Shivprasad koirala**

11 Oct 2011 [CPOL](#)

Rate me:  4.78/5 (102 votes)

Interpreter, Iterator, Mediator, Memento and Observer Pattern

## Introduction

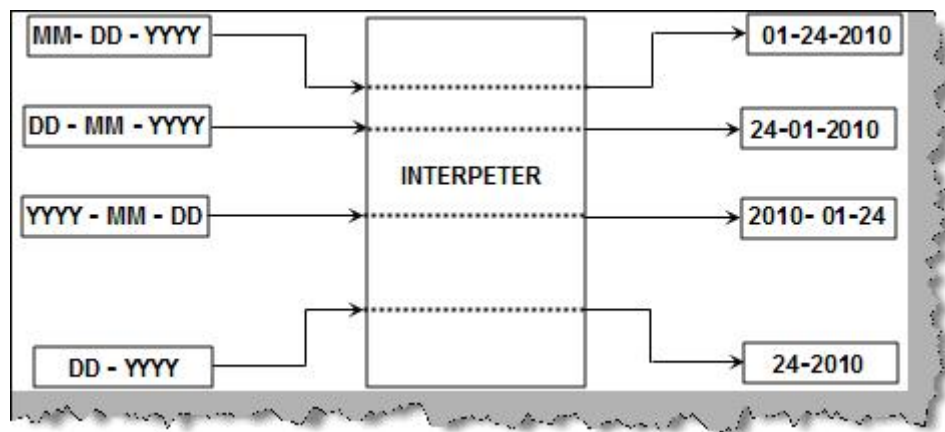
This is continuation to design pattern FAQ Part 1. In this FAQ series we will cover Interpreter , Iterator, mediator, memento and observer design pattern

If you have not read my previous section you can always read from below

- Part 1 Design pattern FAQ's -- [factory pattern](#), [abstract factory pattern](#), [builder pattern](#), [prototype pattern](#), [singleton pattern](#) and [command pattern](#)
- Part 3 Design Pattern FAQ's -- [state pattern](#), [strategy pattern](#), [visitor pattern](#), [adapter pattern](#) and [fly weight pattern](#)
- Part 4 Design Pattern FAQ's -- [bridge pattern](#), [composite pattern](#), [decorator pattern](#), [Façade pattern](#), [chain of responsibility\(COR\)](#), [proxy pattern](#) and [template pattern](#)
- UML Part 1 FAQ's [UML Part 1](#)
- UML Part 2 FAQ's [UML part 2](#)
- UML Part 2 FAQ's [UML part 2](#)
- [Design Pattern with a Project](#)

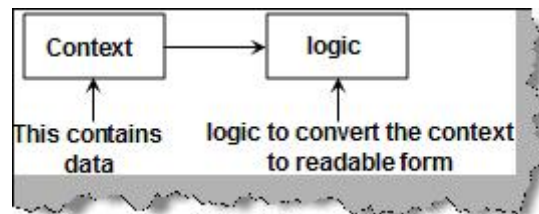
what is Interpreter pattern?

Interpreter pattern allows us to interpret grammar in to code solutions. Ok, what does that mean?. Grammars are mapped to classes to arrive to a solution. For instance  $7 - 2$  can be mapped to 'clsMinus' class. In one line interpreter pattern gives us the solution of how to write an interpreter which can read a grammar and execute the same in the code. For instance below is a simple example where we can give the date format grammar and the interpreter will convert the same in to code solutions and give the desired output.



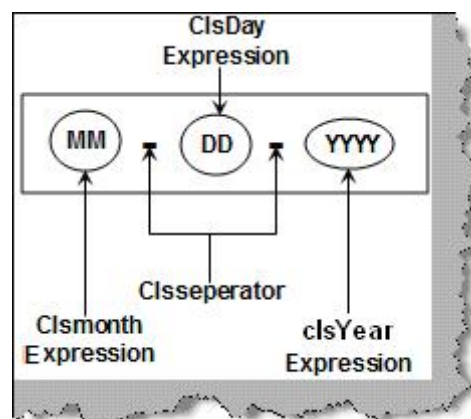
**Figure: - Date Grammar**

Let's make an interpreter for date formats as shown in figure 'Date Grammar'. Before we start let's understand the different components of interpreter pattern and then we will map the same to make the date grammar. Context contains the data and the logic part contains the logic which will convert the context to readable format.



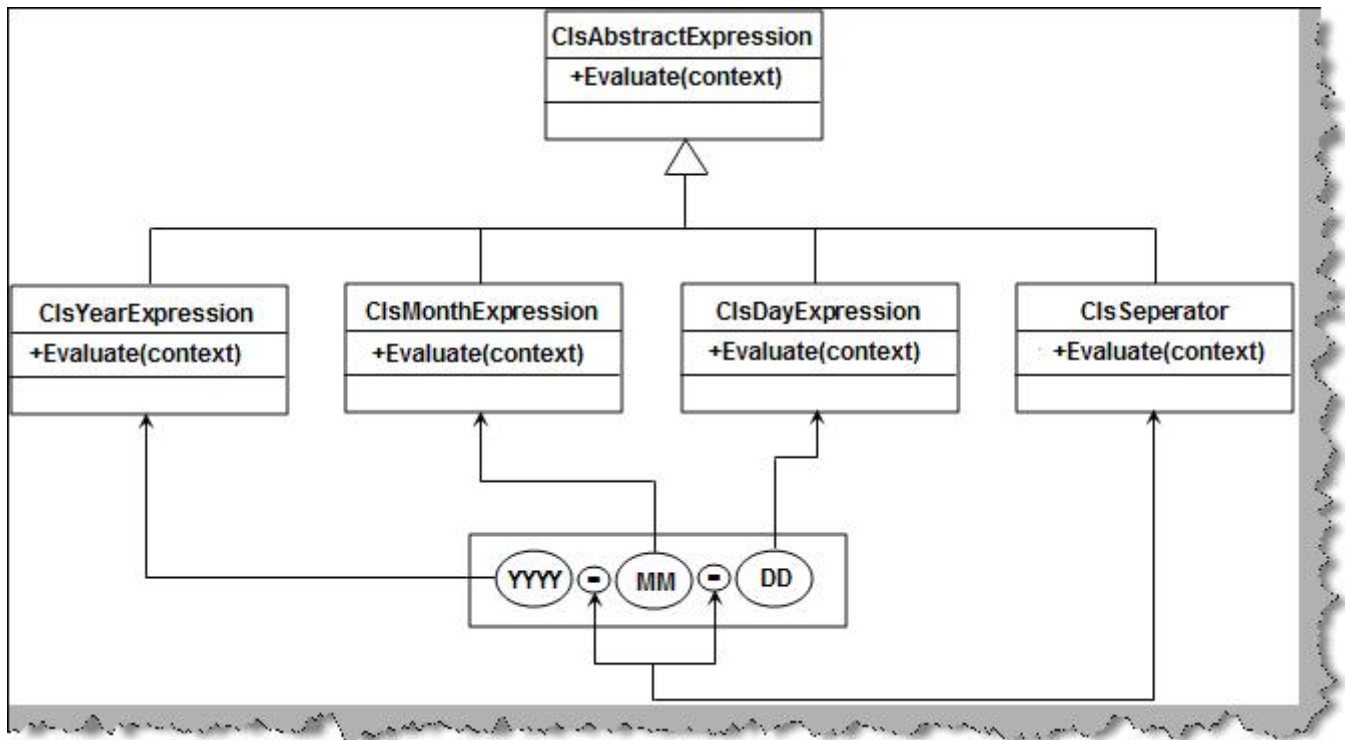
**Figure: - Context and Logic**

Let's understand what is the grammar in the date format is. To define any grammar we should first break grammar in small logical components. Figure 'Grammar mapped to classes' show how different components are identified and then mapped to classes which will have the logic to implement only that portion of the grammar. So we have broken the date format in to four components Month, Day, Year and the separator. For all these four components we will define separate classes which will contain the logic as shown in figure 'Grammar mapped to classes'. So we will be creating different classes for the various components of the date format.



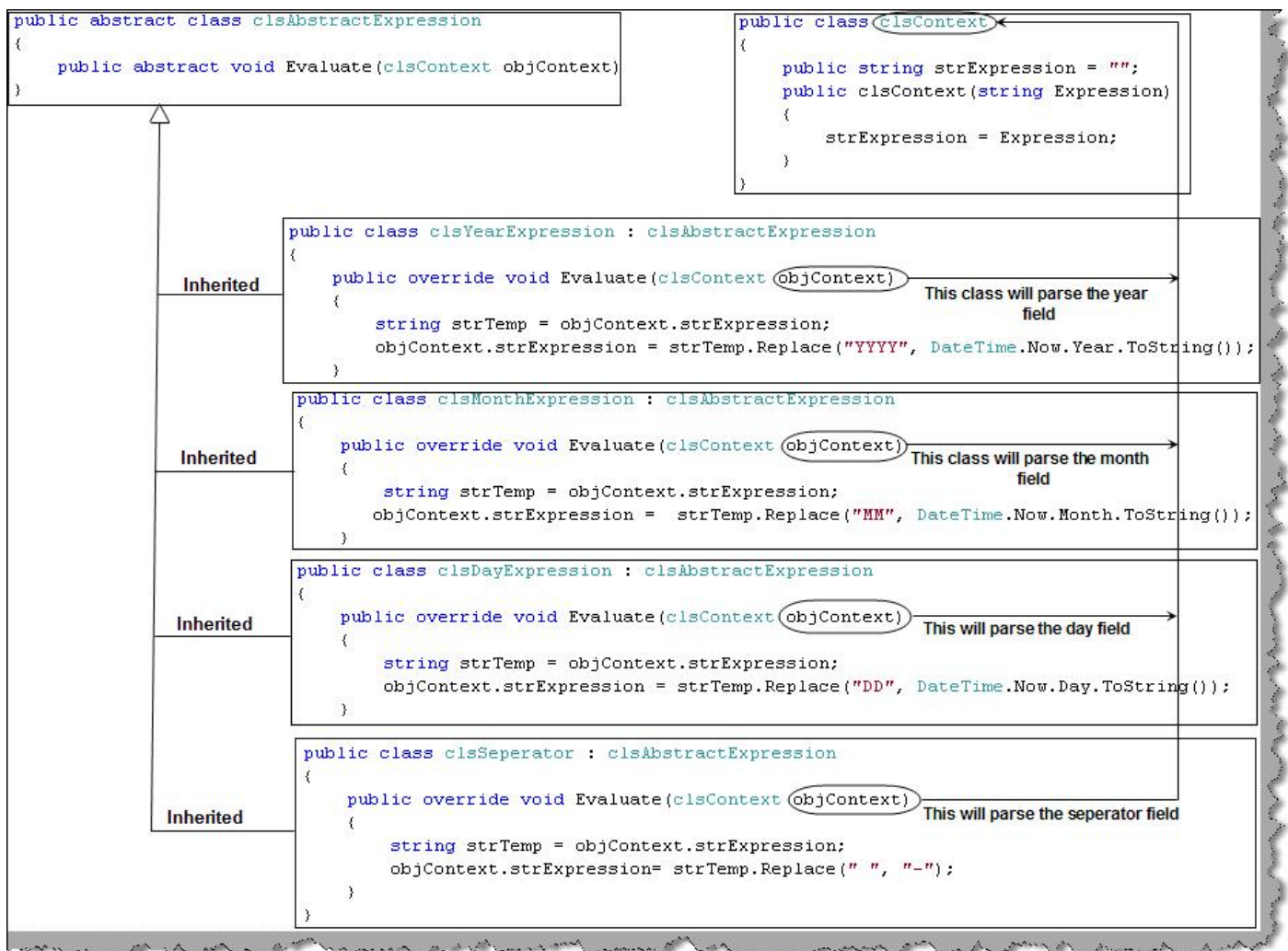
**Figure: - Grammar mapped to classes**

As said there are two classes one is the expression classes which contain logic and the other is the context class which contain data as shown in figure 'Expression and Context classes'. We have defined all the expression parsing in different classes, all these classes inherit from common interface 'ClsAbstractExpression' with a method 'Evaluate'. The 'Evaluate' method takes a context class which has the data; this method parses data according to the expression logic. For instance 'ClsYearExpression' replaces the 'YYYY' with the year value, 'ClsMonthExpression' replaces the 'MM' with month and so on.



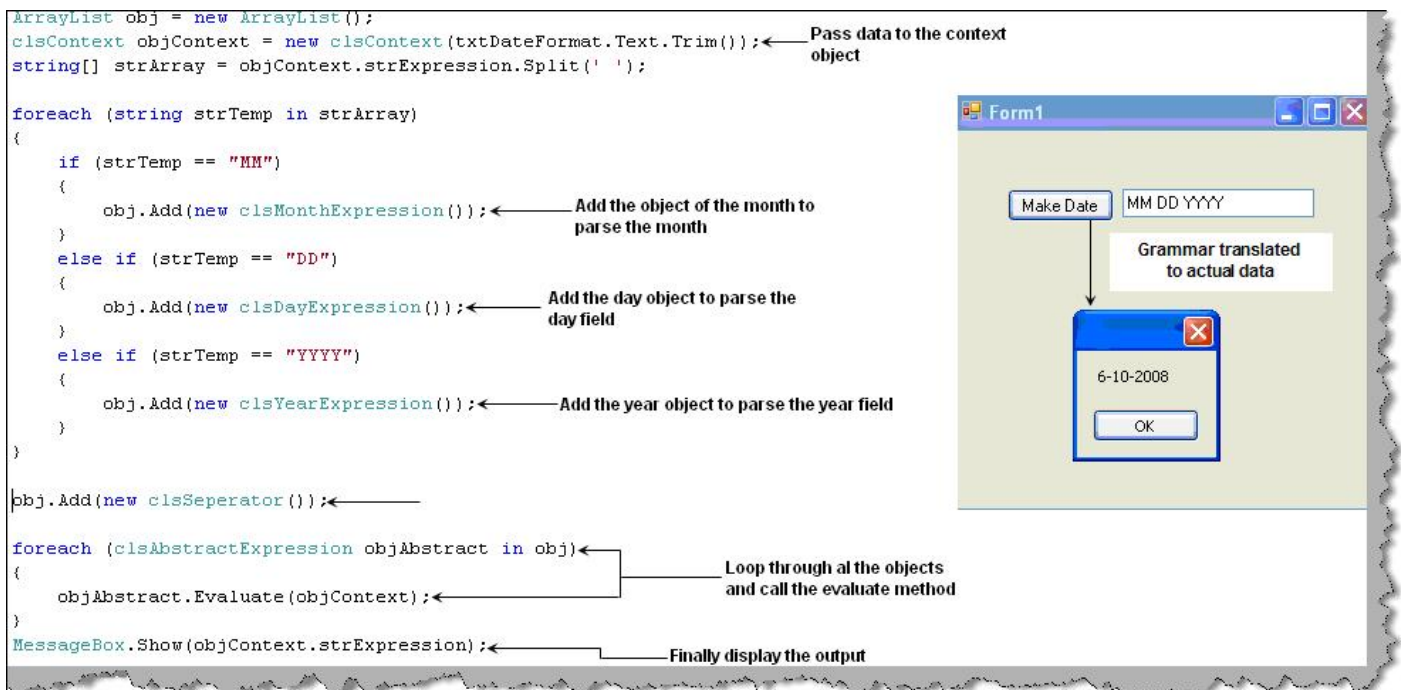
**Figure :- Class diagram for interpreter**





**Figure: - Expression and Context classes**

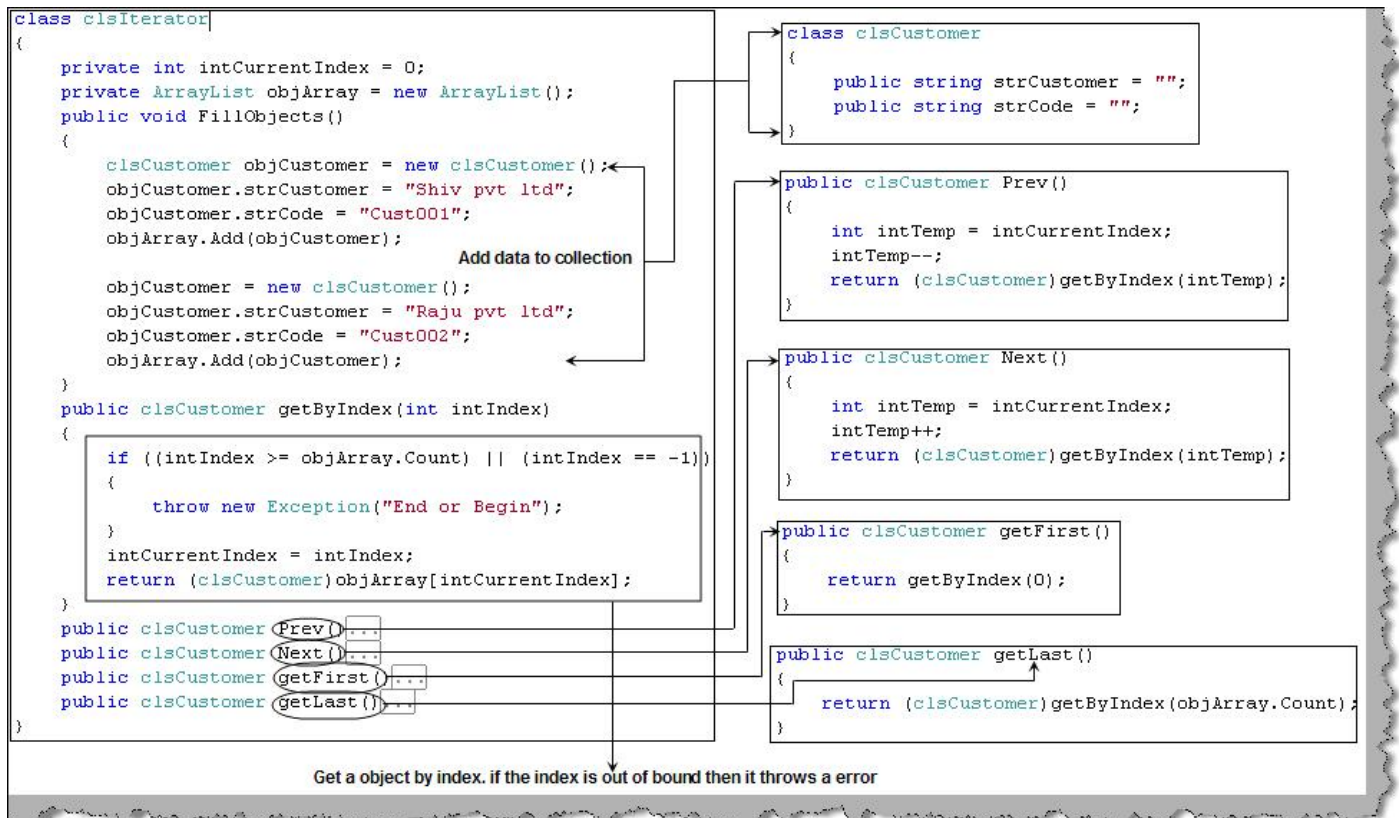
Now that we have separate expression parsing logic in different classes, let's look at how the client will use the iterator logic. The client first passes the date grammar format to the context class. Depending on the date format we now start adding the expressions in a collection. So if we find a 'DD' we add the 'ClsDayExpression', if we find 'MM' we add 'ClsMonthExpression' and so on. Finally we just loop and call the 'Evaluate' method. Once all the evaluate methods are called we display the output.



**Figure: - Client Interpreter logic**

## Can you explain iterator pattern?

Iterator pattern allows sequential access of elements without exposing the inside code. Let's understand what it means. Let's say you have a collection of records which you want to browse sequentially and also maintain the current place which recordset is browsed, then the answer is iterator pattern. It's the most common and unknowingly used pattern. Whenever you use a 'foreach' (It allows us to loop through a collection sequentially) loop you are already using iterator pattern to some extent.

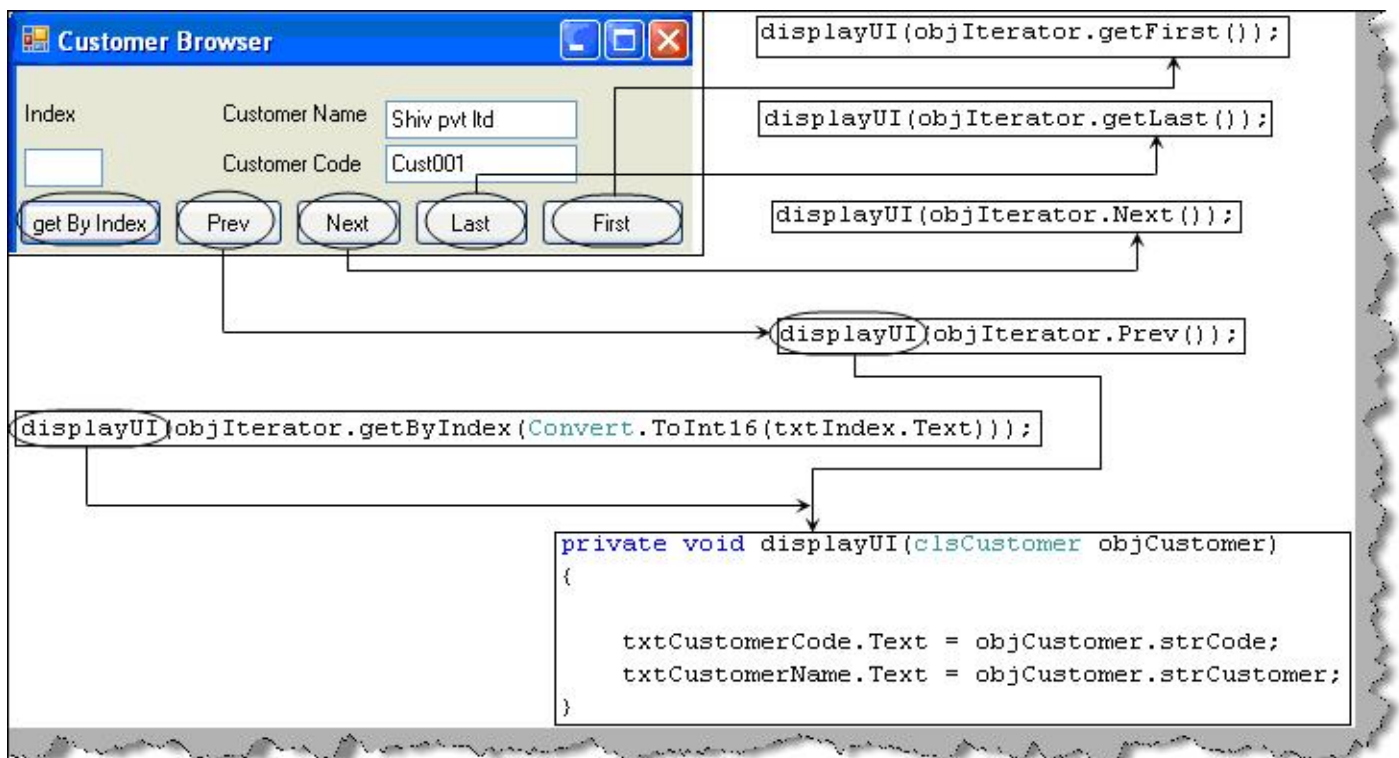


**Figure: - Iterator business logic**

In figure 'Iterator business logic' we have the 'clsIterator' class which has collection of customer classes. So we have defined an array list inside the 'clsIterator' class and a 'FillObjects' method which loads the array list with data. The customer collection array list is private and customer data can be looked up by using the index of the array list. So we have public function like 'getByIndex' (which can look up using a particular index), 'Prev' (Gets the previous customer in the collection), 'Next' (Gets the next customer in the collection), 'getFirst' (Gets the first customer in the collection) and 'getLast' (Gets the last customer in the collection).

So the client is exposed only these functions. These functions take care of accessing the collection sequentially and also it remembers which index is accessed.

Below figures 'Client Iterator Logic' shows how the 'ObjIterator' object which is created from class 'clsIterator' is used to display next, previous, last, first and customer by index.



**Figure: - Client Iterator logic**

Can you explain mediator pattern?

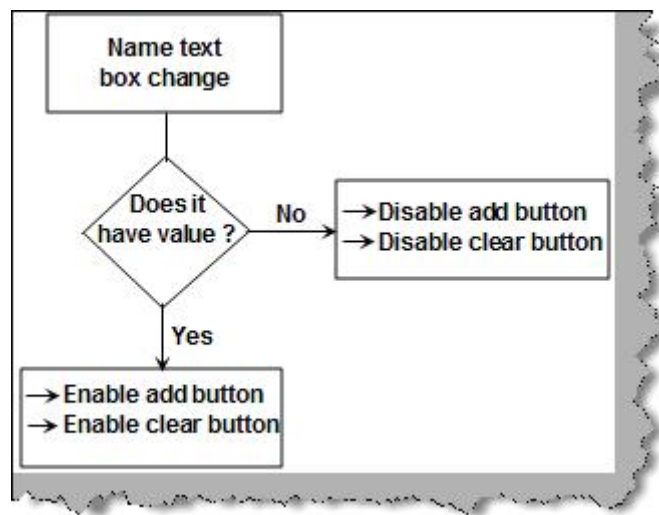
Many a times in projects communication between components are complex. Due to this the logic between the components becomes very complex. Mediator pattern helps the objects to communicate in a disassociated manner, which leads to minimizing complexity.



**Figure: - Mediator sample example**

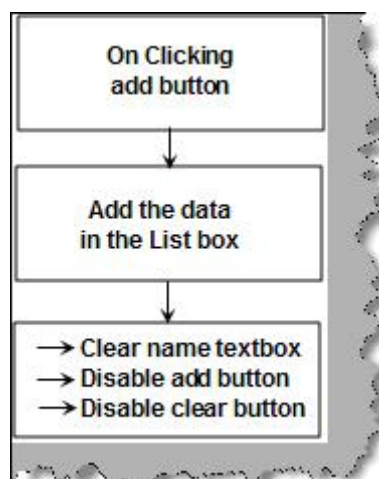
Let's consider the figure 'Mediator sample example' which depicts a true scenario of the need of mediator pattern. It's a very user-friendly user interface. It has three typical scenarios.

Scenario 1:- When a user writes in the text box it should enable the add and the clear button. In case there is nothing in the text box it should disable the add and the clear button.



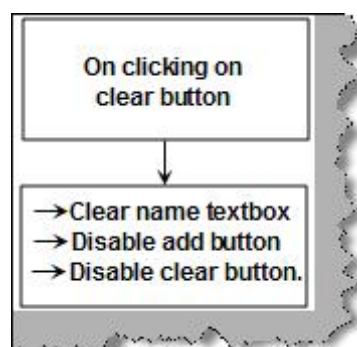
**Figure: - Scenario 1**

Scenario 2:- When the user clicks on the add button the data should get entered in the list box. Once the data is entered in the list box it should clear the text box and disable the add and clear button.



**Figure: - Scenario 2**

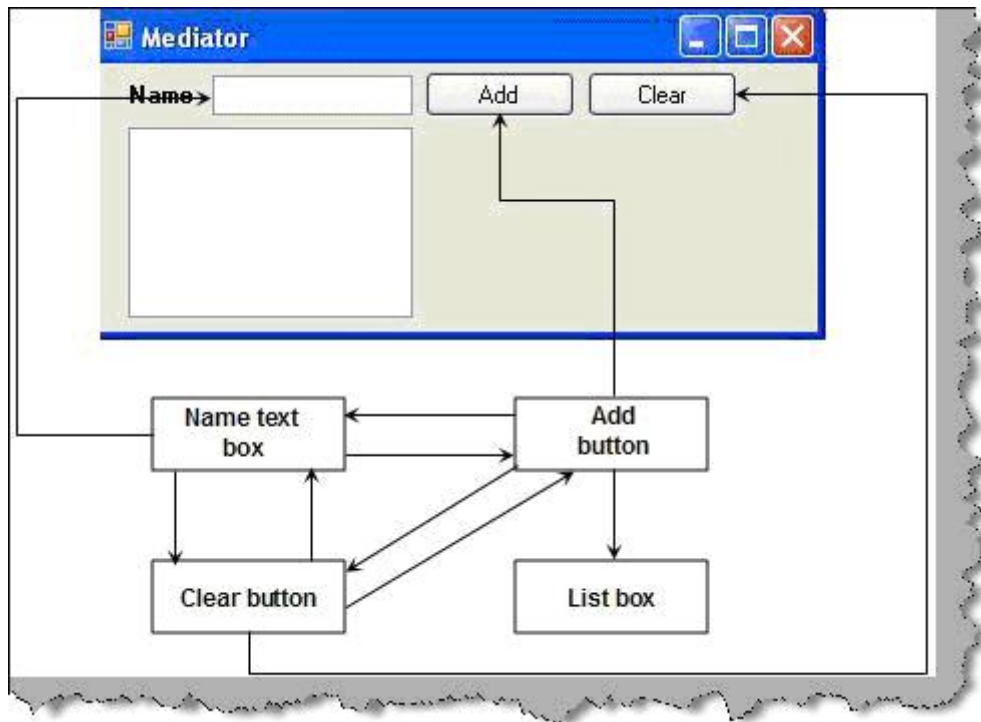
Scenario 3:- If the user click the clear button it should clear the name text box and disable the add and clear button.



**Figure: - Scenario 3**

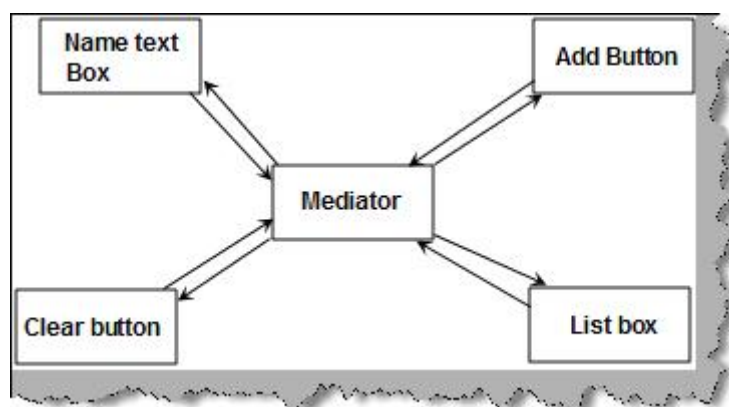


Now looking at the above scenarios for the UI we can conclude how complex the interaction will be in between these UI's. Below figure 'Complex interactions between components' depicts the logical complexity.



**Figure: - Complex interactions between components**

Ok now let me give you a nice picture as shown below 'Simplifying using mediator'. Rather than components communicating directly with each other if they communicate to centralized component like mediator and then mediator takes care of sending those messages to other components, logic will be neat and clean.

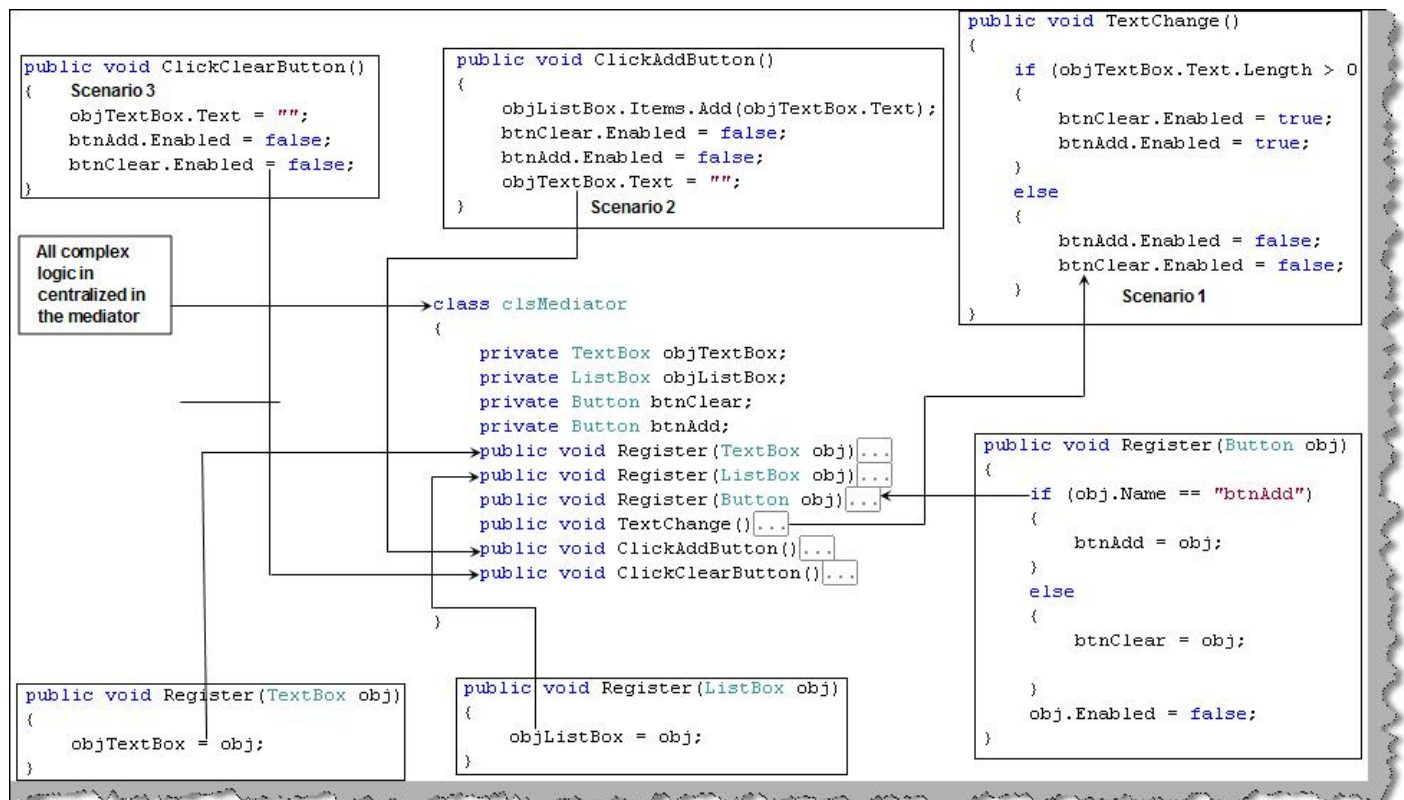


**Figure: - Simplifying using mediator**

Now let's look at how the code will look. We will be using C# but you can easily replicate the thought to JAVA or any other language of your choice. Below figure 'Mediator class' shows the complete code overview of what the mediator class will look like.

The first thing the mediator class does is takes the references of the classes which have the

complex communication. So here we have exposed three overloaded methods by name 'Register'. 'Register' method takes the text box object and the button objects. The interaction scenarios are centralized in 'ClickAddButton', 'TextChange' and 'ClickClearButton' methods. These methods will take care of the enable and disable of UI components according to scenarios.



**Figure: - Mediator class**

The client logic is pretty neat and cool now. In the constructor we first register all the components with complex interactions with the mediator. Now for every scenario we just call the mediator methods. In short when there is a text change we can the 'TextChange' method of the mediator, when the user clicks add we call the 'ClickAddButton' and for clear click we call the 'ClickClearButton'.

```

private clsMediator objMediator = new clsMediator(); ← Create the object of mediator class
public Form1()
{
    InitializeComponent();
    objMediator.Register(txtName); ←
    objMediator.Register(btnAdd); ← Register all the UI components in the mediator
    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(); ← Call the appropriate events in the mediator to handle the complex logic
}

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

```

**Figure: - Mediator client logic**

Can you explain memento pattern?

Memento pattern is the way to capture objects internal state with out violating encapsulation. Memento pattern helps us to store a snapshot which can be reverted at any moment of time by the object. Let's understand what it means in practical sense. Consider figure 'Memento practical example', it shows a customer screen. Let's say if the user starts editing a customer record and he makes some changes. Later he feels that he has done something wrong and he wants to revert back to the original data. This is where memento comes in to play. It will help us store a copy of data and in case the user presses cancel the object restores to its original state.

**Figure: - Memento practical example**

Let's try to complete the same example in C# for the customer UI which we had just gone through.

Below is the customer class 'clsCustomer' which has the aggregated memento class 'clsCustomerMemento' which will hold the snapshot of the data. The memento class 'clsCustomerMemento' is the exact replica (excluding methods) of the customer class 'clsCustomer'. When the customer class 'clsCustomer' gets initialized the memento class also gets initialized. When the customer class data is changed the memento class snapshot is not changed. The 'Revert' method sets back the memento data to the main class.



**Figure: - Customer class for memento**

The client code is pretty simple. We create the customer class. In case we have issues we click the cancel button which in turn calls the 'revert' method and reverts the changed data back to the memento snapshot data. Figure 'Memento client code' shows the same in a pictorial format.



```

clsCustomer objCustomer = new clsCustomer(); ← Customer object created

private void DisplayCustomer() ← Display customer data to the UI
{
    txtCustomerCode.Text=objCustomer.CustomerCode;
    txtCustomerName.Text = objCustomer.CustomerName;
    txtAddress.Text = objCustomer.Address;
}

private void btnUpdate_Click(object sender, EventArgs e) ← Update the customer object
{
    objCustomer.CustomerCode = txtCustomerCode.Text;
    objCustomer.CustomerName = txtCustomerName.Text;
    objCustomer.Address = txtAddress.Text;
}

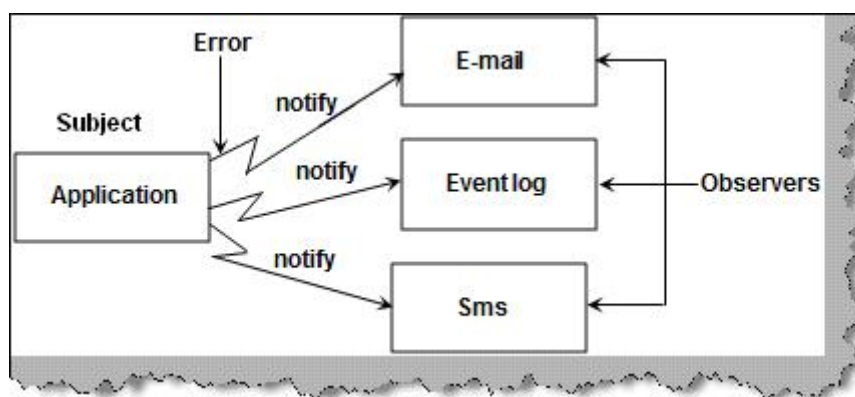
private void btnCancel_Click(object sender, EventArgs e) ← Something has gone wrong so please revert
{
    objCustomer.Revert();
    DisplayCustomer();
}

```

**Figure: - Memento client code**

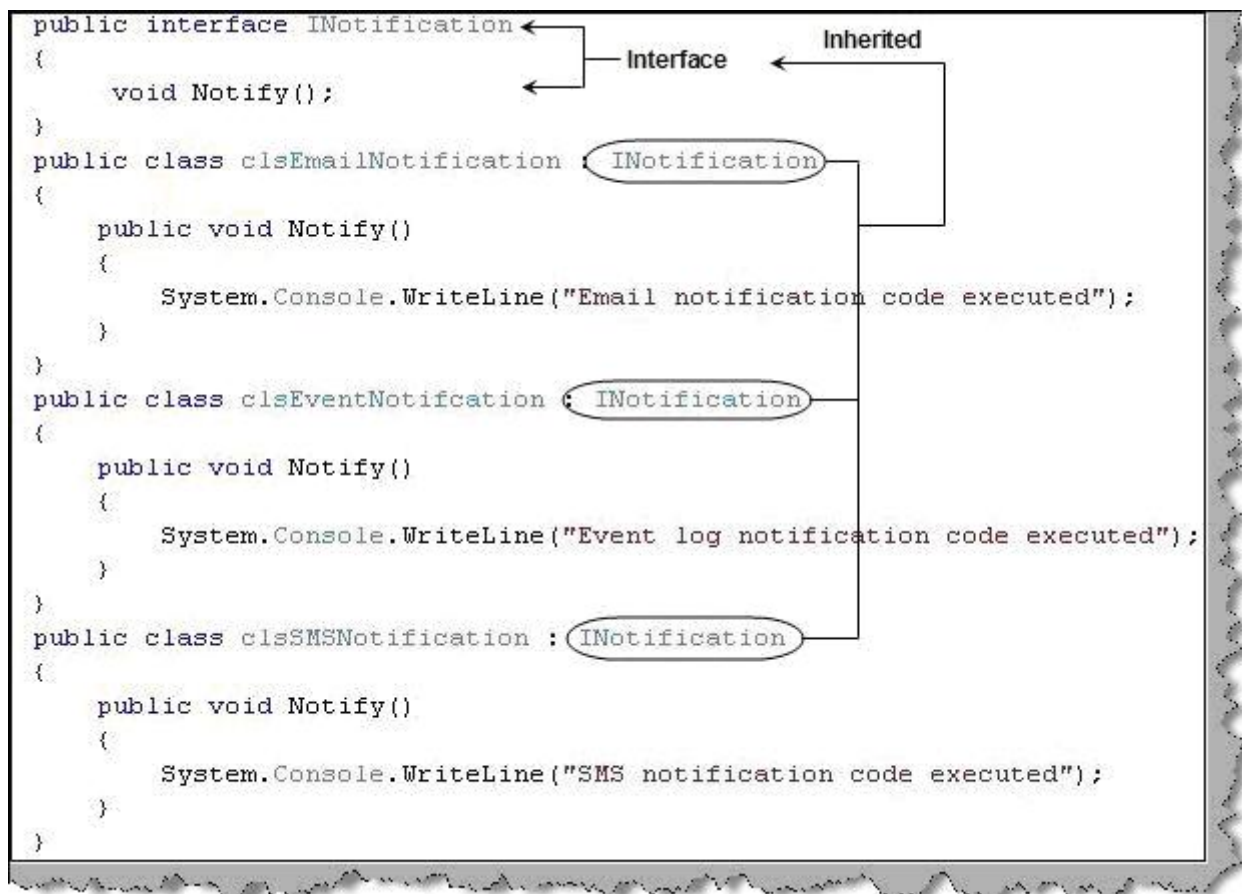
Can you explain observer pattern?

Observer pattern helps us to communicate between parent class and its associated or dependent classes. There are two important concepts in observer pattern 'Subject' and 'Observers'. The subject sends notifications while observers receive notifications if they are registered with the subject. Below figure 'Subject and observers' shows how the application (subject) sends notification to all observers (email, event log and SMS). You can map this example to publisher and subscriber model. The publisher is the application and subscribers are email, event log and sms.



**Figure: - Subject and Observers**

Let's try to code the same example which we have defined in the previous section. First let's have a look at the subscribers / notification classes. Figure 'Subscriber classes' shows the same in a pictorial format. So we have a common interface for all subscribers i.e. 'INotification' which has a 'notify' method. This interface 'INotification' is implemented by all concrete notification classes. All concrete notification classes define their own notification methodology. For the current scenario we have just displayed a print saying the particular notification is executed.



**Figure: - Subscriber classes**

As said previously there are two sections in an observer pattern one is the observer/subscriber which we have covered in the previous section and second is the publisher or the subject.

The publisher has a collection of arraylist which will have all subscribers added who are interested in receiving the notifications. Using 'addNotification' and 'removeNotification' we can add and remove the subscribers from the arraylist. 'NotifyAll' method loops through all the subscribers and send the notification.



```

public class clsNotifier
{
    private ArrayList objNotifications = new ArrayList(); ← Will contain all the
                                                         notification objects

    public void addNotification(INotification obj)
    {
        objNotifications.Add(obj); ← Add the subscribers i.e email,sms and eventlogs
    }
    public void removeNotification(INotification obj)
    {
        objNotifications.Remove(obj); ← Remove the notifications
    }
    public void NotifyAll()
    {
        foreach (INotification objNotification in objNotifications)
        {
            objNotification.Notify(); ← Notify all the subscribers
                                     which are registered
        }
    }
}

```

**Figure: - Publisher/Subject classes**

Now that we have an idea about the publisher and subscriber classes lets code the client and see observer in action. Below is a code for observer client snippet. So first we create the object of the notifier which has collection of subscriber objects. We add all the subscribers who are needed to be notified in the collection.

Now if the customer code length is above 10 characters then tell notify all the subscribers about the same.

```

// 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(); ← Create a object of notifier

// Add subjects/subscribers which needs to be notified
clsEmailNotification objEmailNotification = new clsEmailNotification();
clsEventNotification objEventNotification = new clsEventNotification();
objNotifier.addNotification(objEmailNotification);
objNotifier.addNotification(objEventNotification); ← Create the object
                                                         and add the
                                                         subscribers
                                                         to the notifiers

// 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) ← If the customer length is
{                                     more than 20 characters
    objNotifier.NotifyAll(); ← then send notification to all
                             the subscribers/observers
}

Console.ReadLine();

```

## Figure: - Observer client code

# Design pattern FAQ Part 3 ( Design pattern training series)



**Shivprasad koirala**

20 Oct 2011 CPOL

Rate me:  3.69/5 (64 votes)

Design pattern FAQ Part 3 State Pattern, Strategy pattern, Visitor pattern, Adapter and fly weight

## Introduction

This FAQ article is continuation to design pattern FAQ part 1 and 2. In this article we will try to understand state pattern, strategy, visitor, adapter and flyweight pattern.

In case you are completely new to design patterns or you really do not want to read this complete article do see our free [design pattern Training and interview questions / answers](#) videos.

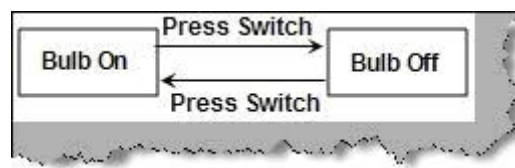
If you have not read my previous section you can always read from below

- Part 1 Design pattern FAQ's -- [factory pattern](#), [abstract factory pattern](#), [builder pattern](#), [prototype pattern](#), [singleton pattern](#) and [command pattern](#)
- Part 2 Design Pattern FAQ's -- [Interpreter pattern](#), [iterator pattern](#), [mediator pattern](#), [memento pattern](#) and [observer pattern](#)
- Part 4 Design Pattern FAQ's -- [bridge pattern](#), [composite pattern](#), [decorator pattern](#), [Façade pattern](#), [chain of responsibility\(COR\)](#), [proxy pattern](#) and [template pattern](#)
- UML Part 1 Interview questions [UML Part 1](#)
- UML Part 2 interview questions [UML part 2](#)
- [Design Pattern with a Project](#)

Can you explain state pattern?

State pattern allows an object to change its behavior depending on the current values of the object. Consider the figure 'State pattern example'. It's an example of a bulb operation. If the state

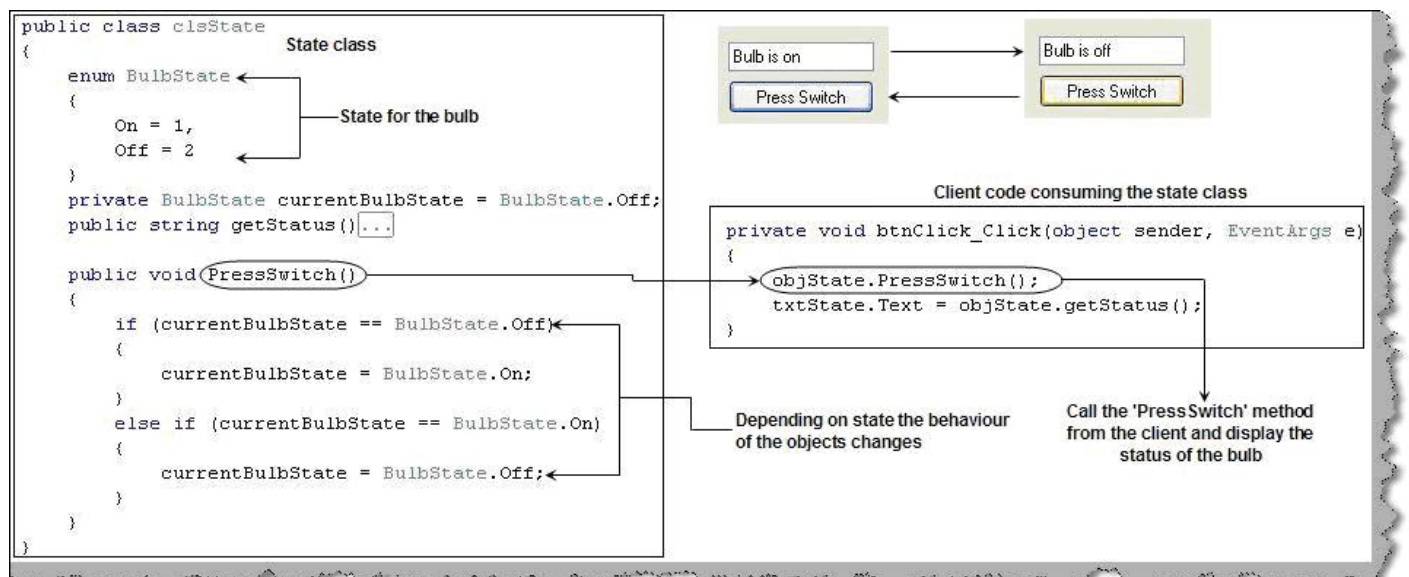
of the bulb is off and you press the switch the bulb will turn on. If the state of bulb is on and you press the switch the bulb will be off. So in short depending on the state the behavior changes.



**Figure: - State pattern example**

Now let's try to implement the same bulb sample in C#. Figure 'State pattern in action' shows both the class and the client code. We have made a class called as 'clsState' which has an enum with two state constants 'On' and 'Off'. We have defined a method 'PressSwitch' which toggles its state depending on the current state. In the right hand side of the same figure we have defined a client which consumes the 'clsState' class and calls the 'PressSwitch()' method. We have displayed the current status on the textbox using the 'getStatus' function.

When we click the press switch it toggles to the opposite state of what we have currently.

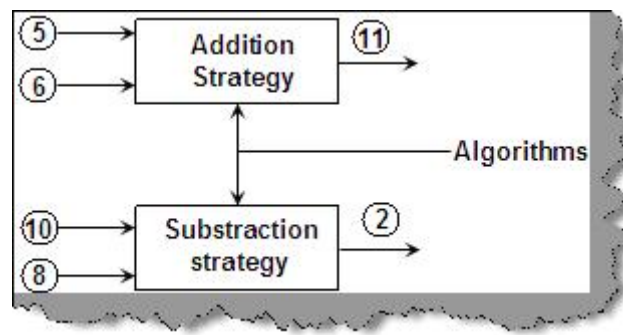


**Figure: - State pattern in action**

Can you explain strategy pattern?

Strategy pattern are algorithms inside a class which can be interchanged depending on the class used. This pattern is useful when you want to decide on runtime which algorithm to be used.

Let's try to see an example of how strategy pattern works practically. Let's take an example of a math's calculation where we have strategies like add and subtract. Figure 'Strategy in action' shows the same in a pictorial format. It takes two numbers and the depending on the strategy it gives out results. So if it's an addition strategy it will add the numbers, if it's a subtraction strategy it will give the subtracted results. These strategies are nothing but algorithms. Strategy pattern are nothing but encapsulation of algorithms inside classes.



**Figure: - Strategy in action**

So the first thing we need to look in to is how these algorithms can be encapsulated inside the classes. Below figure 'Algorithm encapsulated' shows how the 'add' is encapsulated in the 'clsAddStategy' class and 'subtract' in the 'clsSubtractStategy' class. Both these classes inherit from 'clsStrategy' defining a 'calculate' method for its child classes.

### **Figure: - Algorithms encapsulated**

Now we define a wrapper class called as 'clsMaths' which has a reference to the 'clsStategy' class. This class has a 'setStategy' method which sets the strategy to be used.



**Figure: - Strategy and the wrapper class**

Below figure 'Strategy client code' shows how the wrapper class is used and the strategy object is set on runtime using the 'setStrategy' method.

### **Figure: - Strategy client code**

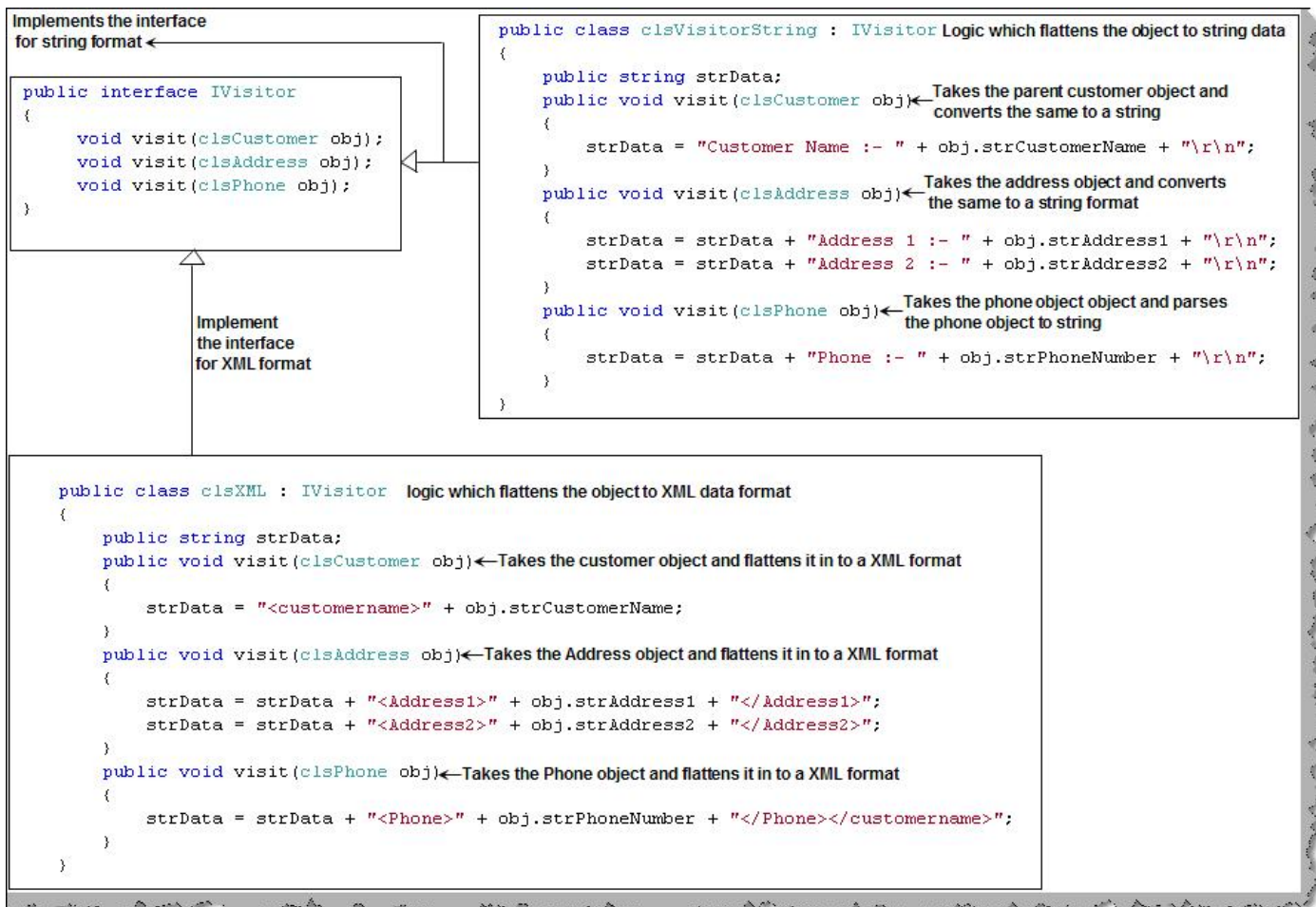
Can you explain visitor pattern?

Visitor pattern allows us to change the class structure with out changing the actual class. Its way of separating the logic and algorithm from the current data structure. Due to this you can add new logic to the current data structure with out altering the structure. Second you can alter the structure with out touching the logic.

Consider the below figure 'Logic and data structure' where we have a customer data structure. Every customer object has multiple address objects and every address object had multiple phone objects. This data structure needs to be displayed in two different formats one is simple string and second XML. So we have written two classes one is the string logic class and other is the XML logic class. These two classes traverse through the object structure and give the respective outputs. In short the visitor contains the logic.

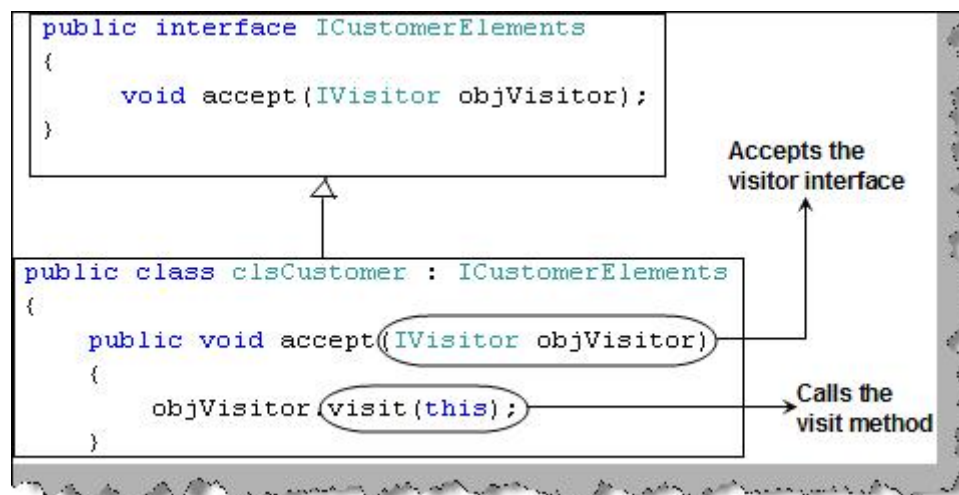
### **Figure: - Logic and data structure**

Let's take the above customer sample and try to implement the same in C#. If you are from other programming you should be able to map the same accordingly. We have created two visitor classes one which will be used to parse for the string logic and other for XML. Both these classes have a visit method which takes each object and parses them accordingly. In order to maintain consistency we have implemented them from a common interface 'IVisitor'.



**Figure :- Visitor class**

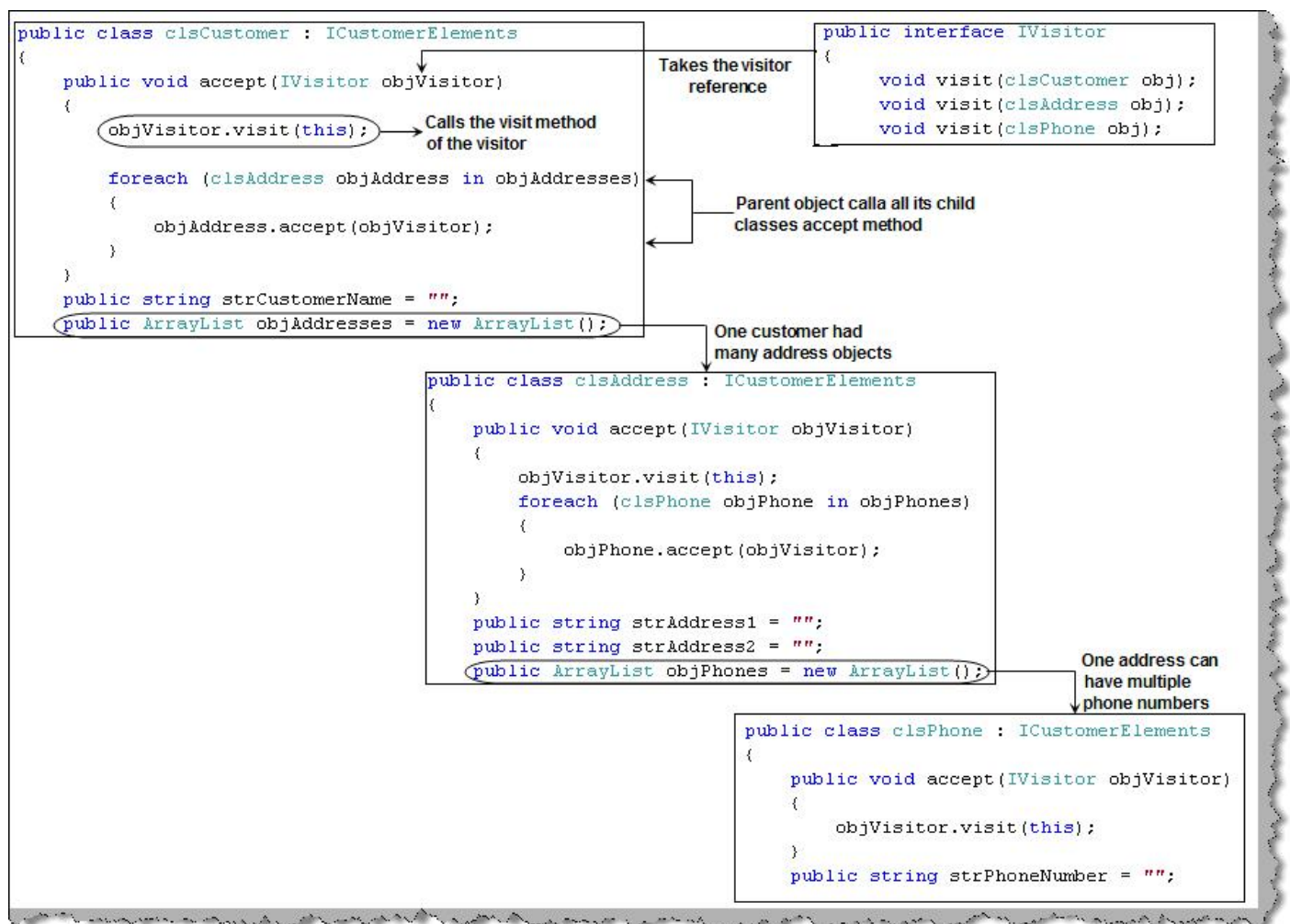
The above defined visitor class will be passed to the data structure class i.e. the customer class. So in the customer class we have passed the visitor class in an 'Accept' function. In the same function we pass this class type and call the visit function. The visit function is overloaded so it will call according to the class type passed.



**Figure: - Visitor passed to data structure class**

Now every customer has multiple address objects and every address has multiple phone objects. So we have 'objAddresses' arraylist object aggregated in the 'clsCustomer' class and 'objPhones'

arraylist aggregated in the 'clsAddress' class. Every object has the accept method which takes the visitor class and passes himself in the visit function of the visitor class. As the visit function of the visitor class is overloaded it will call the appropriate visitor method as per polymorphism.



**Figure: - Customer, Address and phones**

Now that we have the logic in the visitor classes and data structure in the customer classes its time to use the same in the client. Below figure 'Visitor client code' shows a sample code snippet for using the visitor pattern. So we create the visitor object and pass it to the customer data class. If we want to display the customer object structure in a string format we create the 'clsVisitorString' and if we want to generate in XML format we create the 'clsXML' object and pass the same to the customer object data structure. You can easily see how the logic is now separated from the data structure.

**Figure: - Visitor client code**

What the difference between visitor and strategy pattern?

Visitor and strategy look very much similar as they deal with encapsulating complex logic from data. We can say visitor is more general form of strategy.

In strategy we have one context or a single logical data on which multiple algorithms operate. In the previous questions we have explained the fundamentals of strategy and visitor. So let's understand the same by using examples which we have understood previously. In strategy we have a single context and multiple algorithms work on it. Figure 'Strategy' shows how we have a one data context and multiple algorithm work on it.



### **Figure: - Strategy**

In visitor we have multiple contexts and for every context we have an algorithm. If you remember the visitor example we had written parsing logic for every data context i.e. customer, address and phones object.

### **Figure: - Visitor**

So in short strategy is a special kind of visitor. In strategy we have one data context and multiple algorithms while in visitor for every data context we have one algorithm associated. The basic criteria of choosing whether to implement strategy or visitor depends on the relationship between

context and algorithm. If there is one context and multiple algorithms then we go for strategy. If we have multiple contexts and multiple algorithms then we implement visitor algorithm.

Can you explain adapter pattern?

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. Consider the below figure 'Incompatible interfaces' both of them are collections to hold string values. Both of them have a method which helps us to add string in to the collection. One of the methods is named as 'Add' and the other as 'Push'. One of them uses the collection object and the other the stack. We want to make the stack object compatible with the collection object.

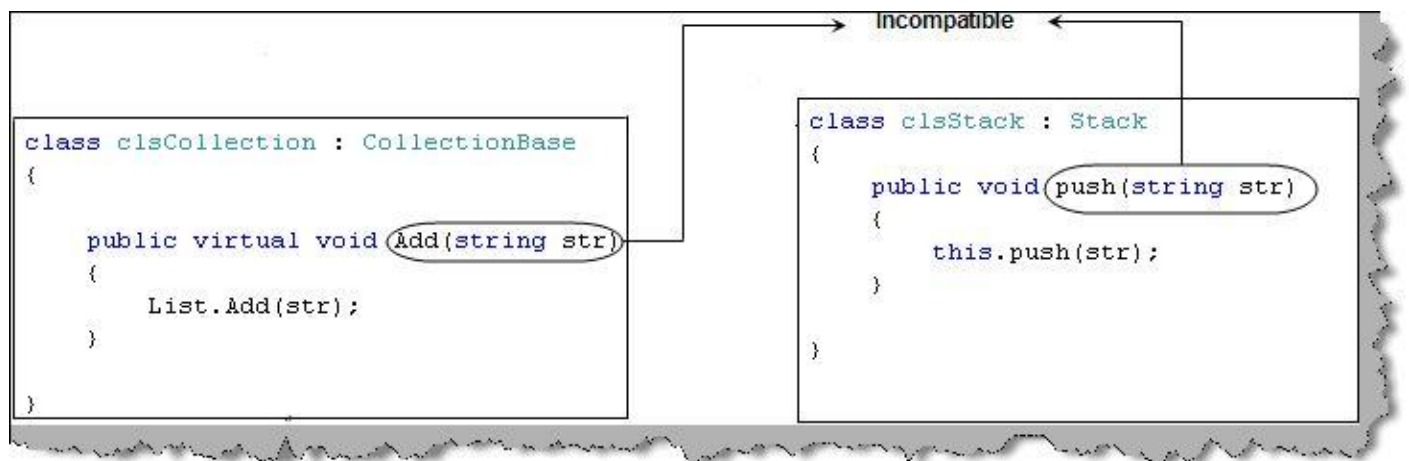
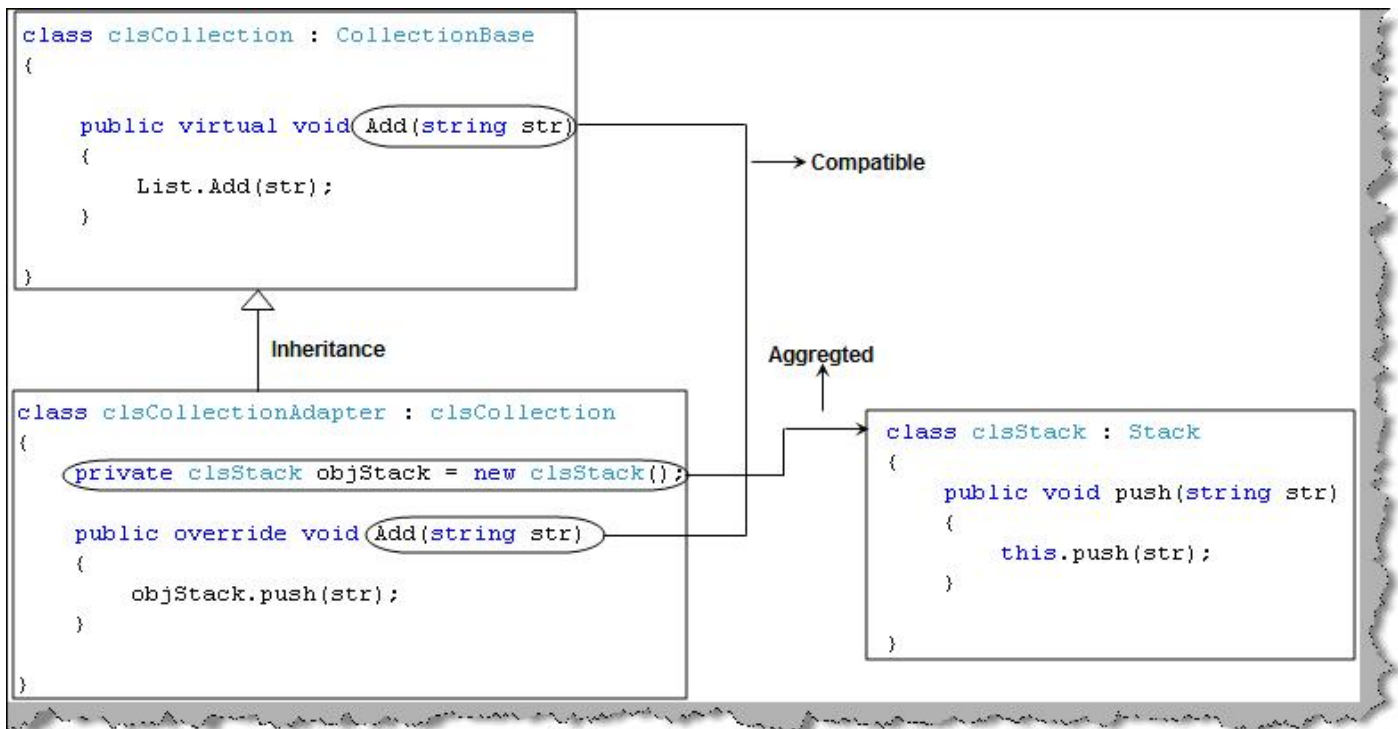


Figure: - Incompatible interfaces

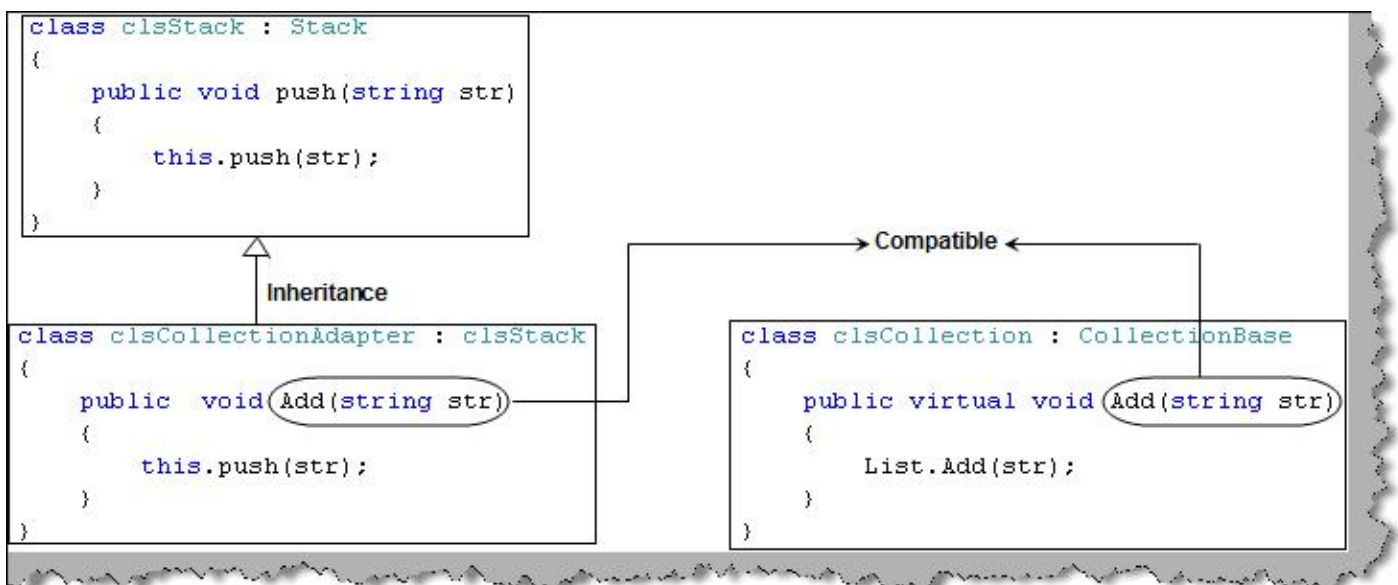
There are two way of implementing adapter pattern one is by using aggregation (this is termed as the object adapter pattern) and the other inheritance (this is termed as the class adapter pattern). First let's try to cover object adapter pattern.

Figure 'Object Adapter pattern' shows a broader view of how we can achieve the same. We have a introduced a new wrapper class '`clsCollectionAdapter`' which wraps on the top of the '`clsStack`' class and aggregates the '`push`' method inside a new '`Add`' method, thus making both the classes compatible.



**Figure: - Object Adapter pattern**

The other way to implement the adapter pattern is by using inheritance also termed as class adapter pattern. Figure 'Class adapter pattern' shows how we have inherited the 'clsStack' class in the 'clsCollectionAdapter' and made it compatible with the 'clsCollection' class.

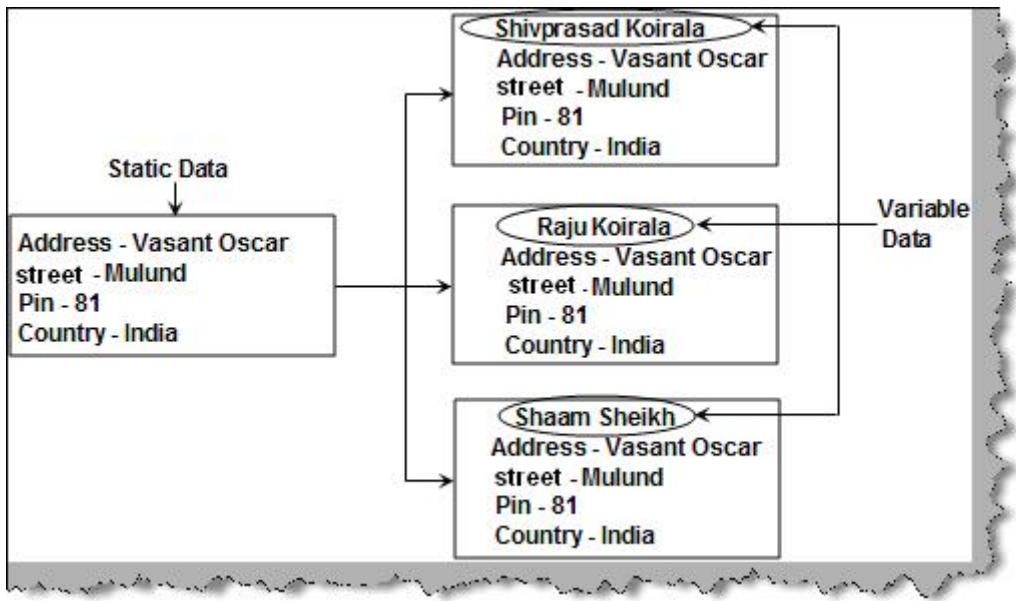


**Figure :- Class adapter pattern**

What is fly weight pattern?

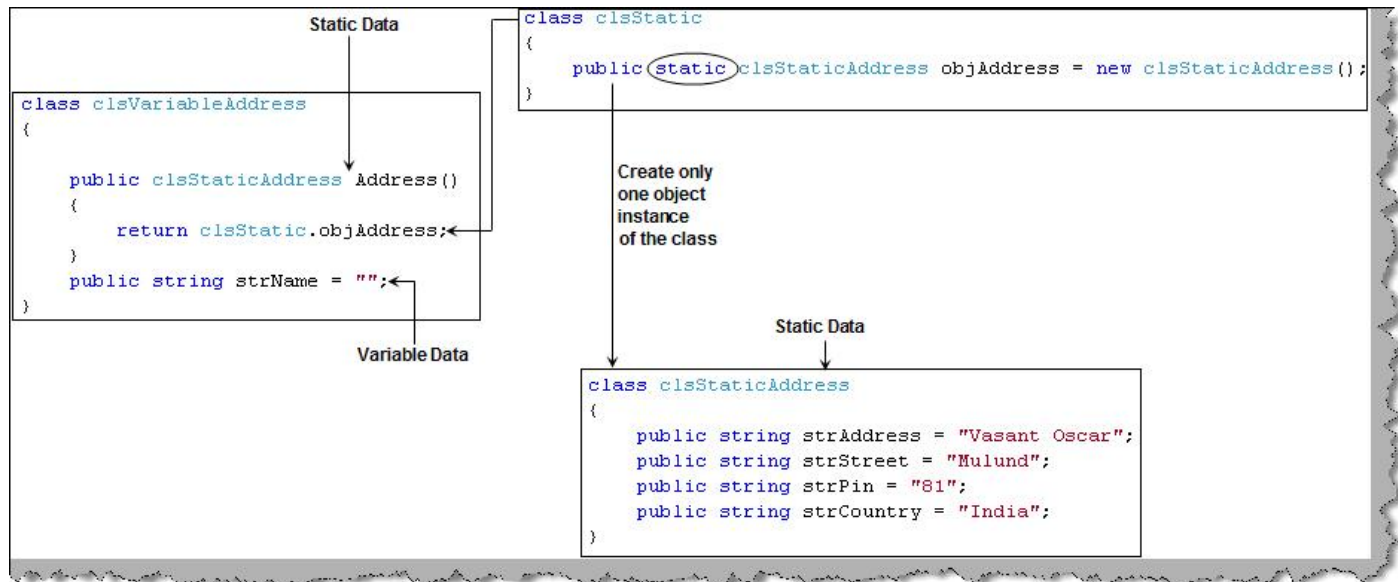
Fly weight pattern is useful where we need to create many objects and all these objects share some kind of common data. Consider figure 'Objects and common data'. We need to print visiting card for all employees in the organization. So we have two parts of data one is the variable data i.e. the employee name and the other is static data i.e. address. We can minimize memory by just keeping one copy of the static data and referencing the same data in all objects of variable data. So we

create different copies of variable data, but reference the same copy of static data. With this we can optimally use the memory.



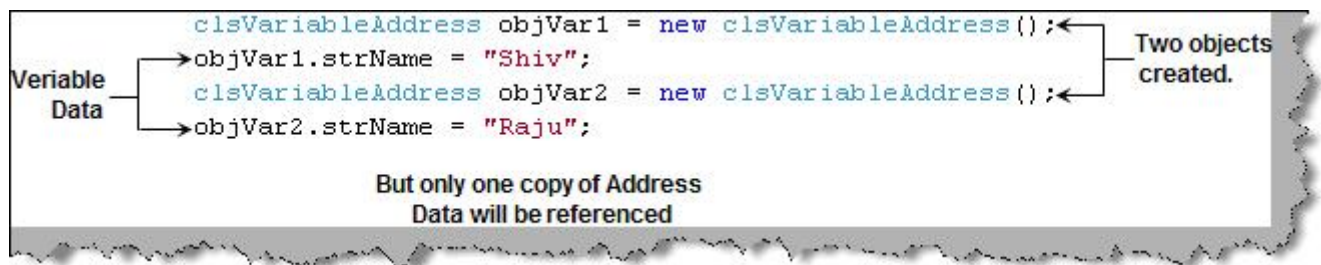
**Figure: - Objects and common data**

Below is a sample C# code demonstration of how flyweight can be implemented practically. We have two classes, 'clsVariableAddress' which has the variable data and second 'clsAddress' which has the static data. To ensure that we have only one instance of 'clsAddress' we have made a wrapper class 'clsStatic' and created a static instance of the 'clsAddress' class. This object is aggregated in the 'clsVariableAddress' class.



**Figure: - Class view of flyweight**

Figure 'Fly weight client code' shows we have created two objects of 'clsVariableAddress' class, but internally the static data i.e. the address is referred to only one instance.



**Figure: - Fly weight client code**

In case your are completely new to design patterns or you really do not want to read this complete article do see our free [design pattern Training and interview questions / answers](#) videos.

Check out factory design pattern video [click here](#)

# Design pattern FAQ part 4



**Shivprasad koirala**  
27 Mar 2011 CPOL

Rate me: ★★★★★ 3.84/5 (70 votes)

Bridge Pattern, Composite Pattern, Facade Pattern, Chain Of Responsibility, Proxy Pattern, Template pattern

Update done on explanation of Decorator Pattern, Composite Pattern and Template Pattern

## Design Pattern FAQ Part 4

### Introduction

This FAQ article is continuation to design pattern FAQ part 1 ,2 and 3 . In this article we will try to understand Bridge Pattern, Composite Pattern, Facade Pattern, Chain Of Responsibility, Proxy Pattern and Template pattern.

If you have not read my previous section you can always read from below

Part 1 Design pattern FAQ's -- [factory pattern](#), [abstract factory pattern](#), [builder pattern](#), [prototype pattern](#), [singleton pattern](#) and [command pattern](#)

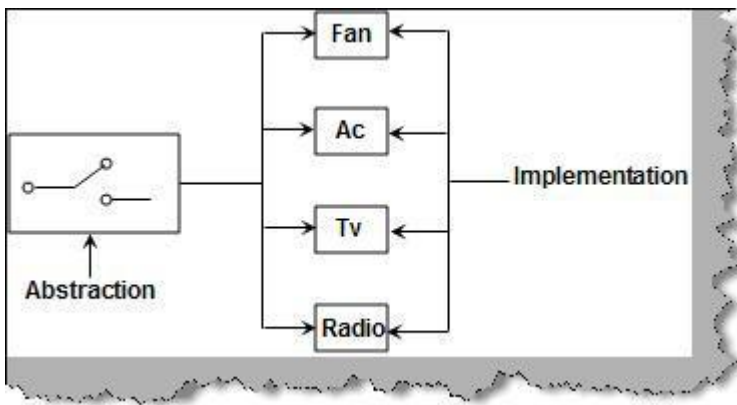
Part 2 Design Pattern FAQ's -- [Interpreter pattern](#), [iterator pattern](#), [mediator pattern](#), [memento pattern](#) and [observer pattern](#)

Part 3 Design Pattern FAQ's -- [state pattern](#), [strategy pattern](#), [visitor pattern](#), [adapter pattern](#) and [fly weight pattern](#)

Can you explain bridge pattern?

Bridge pattern helps to decouple abstraction from implementation. With this if the implementation changes it does not affect abstraction and vice versa. Consider the figure 'Abstraction and Implementation'. The switch is the abstraction and the electronic equipments are the implementations. The switch can be applied to any electronic equipment, so the switch is an abstract thinking while the equipments are implementations.





**Figure: - Abstraction and Implementation**

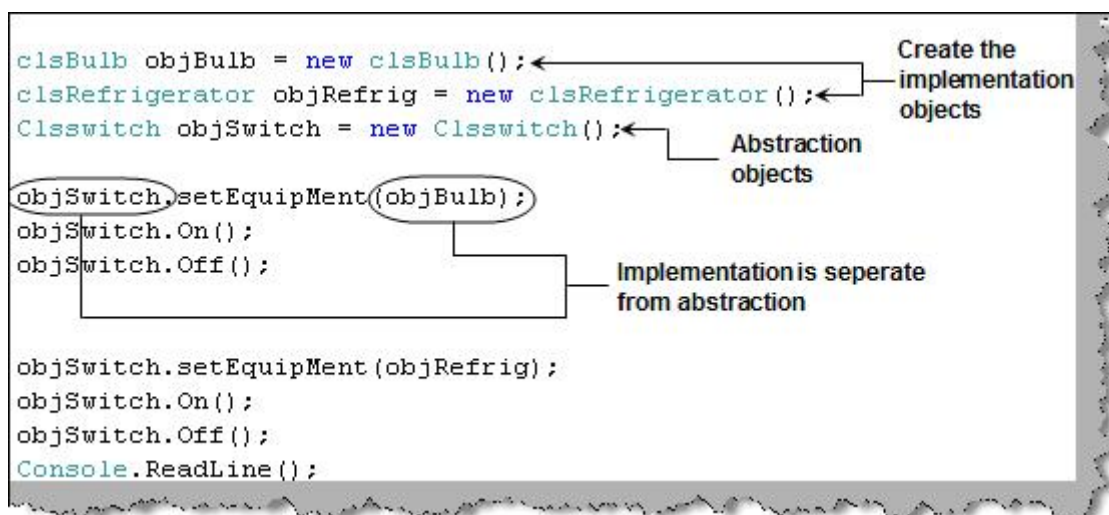
Let's try to code the same switch and equipment example. First thing is we segregate the implementation and abstraction into two different classes. Figure 'Implementation' shows how we have made an interface 'IEquipment' with 'Start()' and 'Stop()' methods. We have implemented two equipments one is the refrigerator and the other is the bulb.

***Figure :- Implementation***

The second part is the abstraction. Switch is the abstraction in our example. It has a 'SetEquipment' method which sets the object. The 'On' method calls the 'Start' method of the equipment and the 'off' calls the 'stop'.

**Figure: - Abstraction**

Finally we see the client code. You can see we have created the implementation objects and the abstraction objects separately. We can use them in an isolated manner.



**Figure :- Client code using bridge**

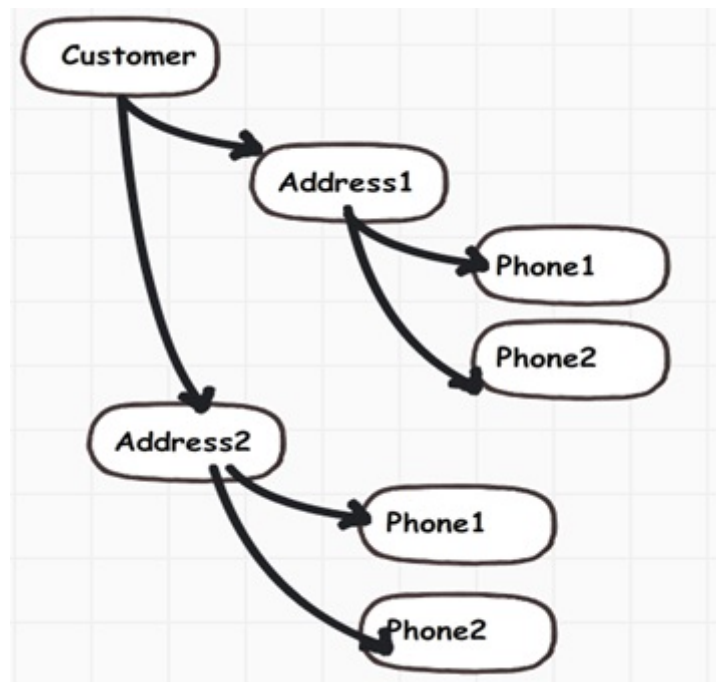
Can you explain composite pattern?



GOF definition :- A tree structure of simple and composite objects

Many times objects are organized in tree structure and developers have to understand the difference between leaf and branch objects. This makes the code more complex and can lead to errors.

For example below is a simple object tree structure where the customer is the main object which has many address objects and every address object references lot of phone objects.



**Figure: - General Process**

Now let's say you want to insert the complete object tree. The sample code will be something as shown below. The code loops through all the customers, all addresses inside the customer object and all phones inside the address objects. While this loop happens the respective update methods are called as shown in the below code snippet.

```
foreach (Customer objCust in objCustomers)
{
objCust.UpDateCustomer();
    foreach (Address oAdd in objCust.Addresses)
    {
        oAdd.UpdateAddress();
    }
    foreach (Phone ophone in oAdd.Phones)
    {
        ophone.UpDatePhone();
    }
}
```

The problem with the above code is that the update vocabulary changes for each object. For customer its 'UpdateCustomer' , for address its 'UpdateAddress' and for phone it is 'UpdatePhone'. In other words the main object and the contained leaf nodes are treated differently. This can lead to confusion and make your application error prone.

The code can be cleaner and neat if we are able to treat the main and leaf object uniformly. You

can see in the below code we have created an interface (IBusinessObject) which forces all the classes i.e. customer, address and phone to use a common interface. Due to the common interface all the object now have the method name as "Update".



```
foreach (IBusinessObject ICust in objCustomers)
{
    ICust.Update();
    foreach (IBusinessObject Iaddress in ((Customer)(ICust)).ChildObjects)
    {
        Iaddress.Update();
        foreach (IBusinessObject iphone in ((Address)(Iaddress)).ChildObjects)
        {
            iphone.Update();
        }
    }
}
```

In order to implement composite pattern first create an interface as shown in the below code snippet.



```
public interface IBusinessObject
{
    void Update();
    bool isValid();
    void Add(object o);
}
```

Force this interface across all the root objects and leaf / node objects as shown below.



```
public class Customer : IBusinessObject
{
    private List<Address> _Addresses;
    public IEnumerable<Address> ChildObjects
    {
        get
        {
            return (IEnumerable<Address>)_Addresses;
        }
    }
    public void Add(object objAdd)
    {
        _Addresses.Add((Address) objAdd);
    }
    public void Update()
    {
    }
    public bool isValid()
    {
        return true;
    }
}
```

```
}  
}
```

Force the implementation on the address object also.

Shrink ▲ 

```
public class Address : IBusinessObject  
{  
    private List<Phone> _Phones;  
  
    public IEnumerable<Phone> ChildObjects  
    {  
        get  
        {  
            return (IEnumerable<Phone>)_Phones.ToList<object>();  
        }  
    }  
  
    public void Add(object objPhone)  
    {  
        _Phones.Add((Phone)objPhone);  
    }  
  
    public void Update()  
    {  
    }  
  
    public bool isValid()  
    {  
        return true;  
    }  
}
```

Force the implementation on the last node object i.e. phone.



```
public class Phone : IBusinessObject  
{  
    public void Update()  
    {}  
    public bool isValid()  
    {return true;}  
    public void Add(object o)  
    {  
        // no implementaton  
    }  
}
```

Can you explain decorator pattern ?





Punch :- Decorator pattern adds dynamically stacked behavior thus helping us to change the behavior of the **object** on runtime.

There are situations where we would like to add dynamic stacked behavior to a class on runtime. The word stack is an important word to note. For instance consider the below situation where a hotel sells bread meals. They have four important products and the order can be placed using the below combination:-


- Simple bread.
- Bread with Chicken.
- Bread with drinks
- Bread with chicken and drinks.

In other words the order process behavior and the cost of the order changes on runtime depending on the type of the combination.

### Figure: -


Below is a simple order with only bread which has two functions 'prepare' and 'calculatecost'. We would like to add new products to this basic bread order dynamically on runtime depending on what the customer wants.

Below is a simple interface which every order will have i.e. Prepare and CalculateCost.



```
interface IOrder
{
    string Prepare();
    double CalculateCost();
}
```

The base product is the bread which implements the IOrder interface. We would like to add new products to the bread order and change the behavior of the complete order.



```

public class OrderBread : IOrder
{
    public string Prepare()
    {
        string strPrepare="";
        strPrepare = "Bake the bread in oven\n";
        strPrepare = strPrepare + "Serve the bread";
        return strPrepare;
    }

    public double CalculateCost()
    {
        return 200.30;
    }
}

```

We can alter the bread order dynamically using decorator pattern. To implement decorator pattern is a 5 steps process.

Step1:- Create a decorator class which aggregates the object / interface for which we need to add the behavior dynamically.

```

abstract class OrderDecorator : IOrder
{
    protected IOrder Order;
    . . . . .
    . . . . .
    . . . . .
}

```

This decorator class will house the object and any method calls to the main object will first invoke all the housed objects and then the main object.

So for instance if you are calling the prepare method, this decorator class will invoke all the prepare methods of the housed object and then the final prepare method. You can see how the output changes when decorator comes in to picture.

**Figure: -**

Step 2: - The housed object/ interface pointer needs to be initialized. We can do the same by using various means for the sample below we will just expose a simple constructor and pass the object to the constructor to initialize the housed object.



```
abstract class OrderDecorator : IOrder
{
    protected IOrder Order;
    public OrderDecorator(IOrder oOrder)
    {
        Order = oOrder;
    }
    . . . . .
}
```

Step 3: - We will implement the IOrder interface and invoke the house object methods using the virtual methods. You can see we have created virtual methods which invoke the house object methods.



```
abstract class OrderDecorator : IOrder
{
    protected IOrder Order;

    public OrderDecorator(IOrder oOrder)
    {
        Order = oOrder;
    }
    public virtual string Prepare()
    {
        return Order.Prepare();
    }

    public virtual double CalculateCost()
    {
        return Order.CalculateCost();
    }
}
```

Step 4: - We are done with the important step i.e. creating the decorator. Now we need to create dynamic behavior object which can be added to the decorator to change object behavior on runtime.

Below is a simple chicken order which can be added to the bread order to create a different order all together called as chicken + bread order. The chicken order is created by inheriting from the order decorator class.

Any call to this object first invokes custom functionality of order chicken and then invokes the housed object functionality. For instance you can see When prepare function is called it first called prepare chicken functionality and then invokes the prepare functionality of the housed object.

The calculate cost also adds the chicken cost and the invokes the housed order cost to sum up the total.



```
class OrderChicken : OrderDecorator
{
    public OrderChicken(IOrder oOrder) : base(oOrder)
    {
    }

    public override string Prepare()
    {
        return base.Prepare() + PrepareChicken();
    }
    private string PrepareChicken()
    {
        string strPrepare = "";
        strPrepare = "\nGrill the chicken\n";
    }
}
```

```

        strPrepare = strPrepare + "Stuff in the bread";
        return strPrepare;
    }
    public override double CalculateCost()
    {
        return base.CalculateCost() + 300.12;
    }
}

```

Same way we can also prepare order drinks.

Shrink ▲ 

```

class OrderDrinks : OrderDecorator
{
    public OrderDrinks(IOrder oOrder)
        : base(oOrder)
    {
    }
    public OrderDrinks()
    {
    }
    public override string Prepare()
    {
        return base.Prepare() + PrepareDrinks();
    }
    private string PrepareDrinks()
    {
        string strPrepare = "";
        strPrepare = "\nTake the drink from freezer\n";
        strPrepare = strPrepare + "Serve in glass";
        return strPrepare;
    }

    public override double CalculateCost()
    {
        return base.CalculateCost() + 10.12;
    }
}

```

Step 5:- The final step is see the decorator pattern in action. So from the client side you can write something like this to create a bread order.



```

IOrder Order =new OrderBread();
Console.WriteLine(Order.Prepare());
Order.CalculateCost().ToString();

```

Below is how the output will be displayed for the above client call.



```

Order 1 :- Simple Bread menu
Bake the bread in oven
Serve the bread
200.3

```



If you wish to create the order with chicken, drink and bread, the below client code will help you with the same.



```
Order = new OrderDrinks(new OrderChicken(new OrderBread()));  
Order.Prepare();  
Order.CalculateCost().ToString();
```

For the above code below is the output which combines drinks + chicken + bread.



```
Order 2 :- Drinks with chicken and bread  
Bake the bread in oven  
Serve the bread  
Grill the chicken  
Stuff in the bread  
Take the drink from freezer  
Serve in glass  
510.54
```

In other words you can now attach these behaviors to the main object and change the behavior of the object on runtime.

Below are different order combination we can generate , thus altering the behavior of the order dynamically.

Shrink ▲

```
Order 1 :- Simple Bread menu  
Bake the bread in oven  
Serve the bread  
200.3  
  
Order 2 :- Drinks with chicken and bread  
Bake the bread in oven  
Serve the bread  
Grill the chicken  
Stuff in the bread  
Take the drink from freezer  
Serve in glass  
510.54  
  
Order 3 :- Chicken with bread  
Bake the bread in oven  
Serve the bread  
Grill the chicken  
Stuff in the bread  
500.42  
  
Order 4 :- drink with simple bread  
Bake the bread in oven  
Serve the bread  
Take the drink from freezer
```

## (A) Can you explain Façade pattern?

Façade pattern sits on the top of group of subsystems and allows them to communicate in a unified manner.

### **Figure: - Façade and Subsystem**

Figure 'Order Façade' shows a practical implementation of the same. In order to place an order we need to interact with product, payment and invoice classes. So order becomes a façade which unites product, payment and invoice classes.

### **Figure: - Order Facade**

Figure 'façade in action' shows how class 'clsorder' unifies / uses 'clsproduct', 'clsproduct' and 'clsInvoice' to implement 'PlaceOrder' functionality.

**Figure :- Façade in action**

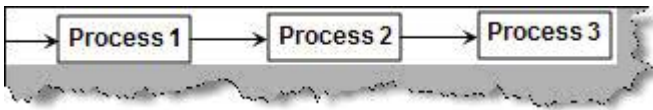
Can you explain chain of responsibility ( COR)?

Chain of responsibility is used when we have series of processing which will be handled by a series of handler logic. Let's understand what that means. There are situations when a request is handled by series of handlers. So the request is taken up by the first handler, he either can handle part of it or can not, once done he passes to the next handler down the chain. This goes on until the proper handler takes it up and completes the processing.



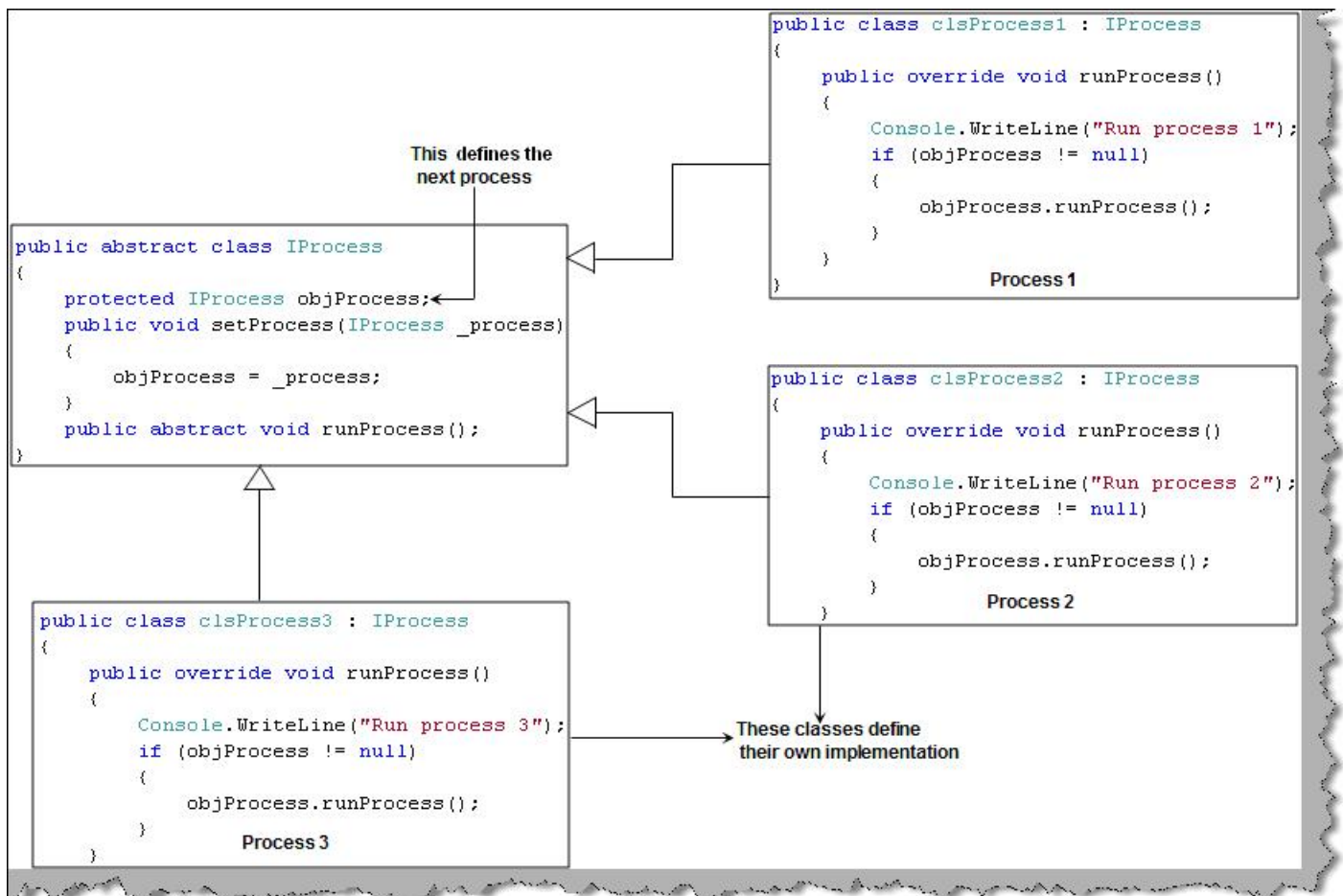
### Figure: - Concept of Chain of Responsibility

Let's try to understand this concept by a small sample example. Consider figure 'Sample example' where we have some logic to be processed. So there are three series of processes which it will go through. So process 1 does some processing and passes the same to process 2. Process 2 does some kind of processing and passed the same to process 3 to complete the processing activity.



### Figure: - Sample example

Figure 'class diagram for COR' the three process classes which inherit from the same abstract class. One of the important points to be noted is that every process points to the next process which will be called. So in the process class we have aggregated one more process object called as 'objProcess'. Object 'ObjProcess' points to next process which should be called after this process is complete.



### Figure: - Class diagram for COR

Now that we have defined our classes its time to call the classes in the client. So we create all the process objects for process1 , process2 and process3. Using the 'setProcess' method we define the link list of process objects. You can see we have set process2 as a link list to process1 and process2 to process3. Once this link list is established we run the process which in turn runs the process according to the defined link list.

```

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

objProcess1.setProcess(objProcess2);
objProcess2.setProcess(objProcess3);

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

```

Diagram annotations:

- Annotations for the first three lines: Create all objects
- Annotations for the next two lines: Set the process link list
- Annotation for the fourth line: Run the process

**Figure: - COR client code**

Can you explain proxy pattern?

Proxy fundamentally is a class functioning as in interface which points towards the actual class which has data. This actual data can be a huge image or an object data which very large and can not be duplicated. So you can create multiple proxies and point towards the huge memory consuming object and perform operations. This avoids duplication of the object and thus saving memory. Proxies are references which points towards the actual object.

Figure 'Proxy and actual object' shows how we have created an interface which is implemented by the actual class. So the interface 'IImageProxy' forms the proxy and the class with implementation i.e. 'clsActualImage' class forms the actual object. You can see in the client code how the interface points towards the actual object.



**Figure: - Proxy and actual object**

The advantages of using proxy are security and avoiding duplicating objects which are of huge sizes. Rather than shipping the code we can ship the proxy, thus avoiding the need of installing the actual code at the client side. With only the proxy at the client end we ensure more security. Second point is when we have huge objects it can be very memory consuming to move to those large objects in a network or some other domain. So rather than moving those large objects we just move the proxy which leads to better performance.

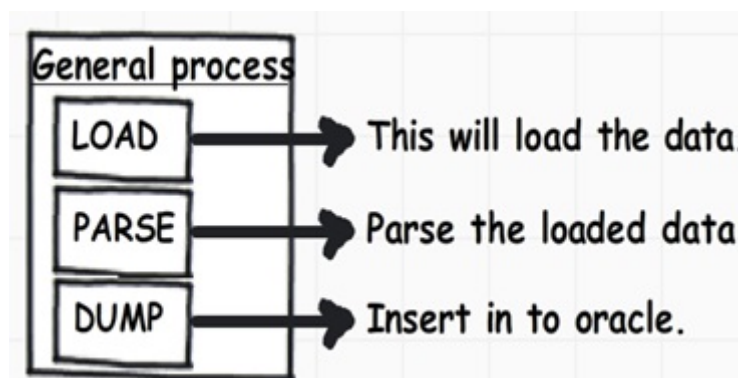
Can you explain template pattern?

Template pattern is a behavioral pattern. Template pattern defines a main process template and this main process template has sub processes and the sequence in which the sub processes can be called. Later the sub processes of the main process can be altered to generate a different behavior.



Punch :- Template pattern is used in scenarios where we want to create extendable behaviors in generalization and specialization relationship.

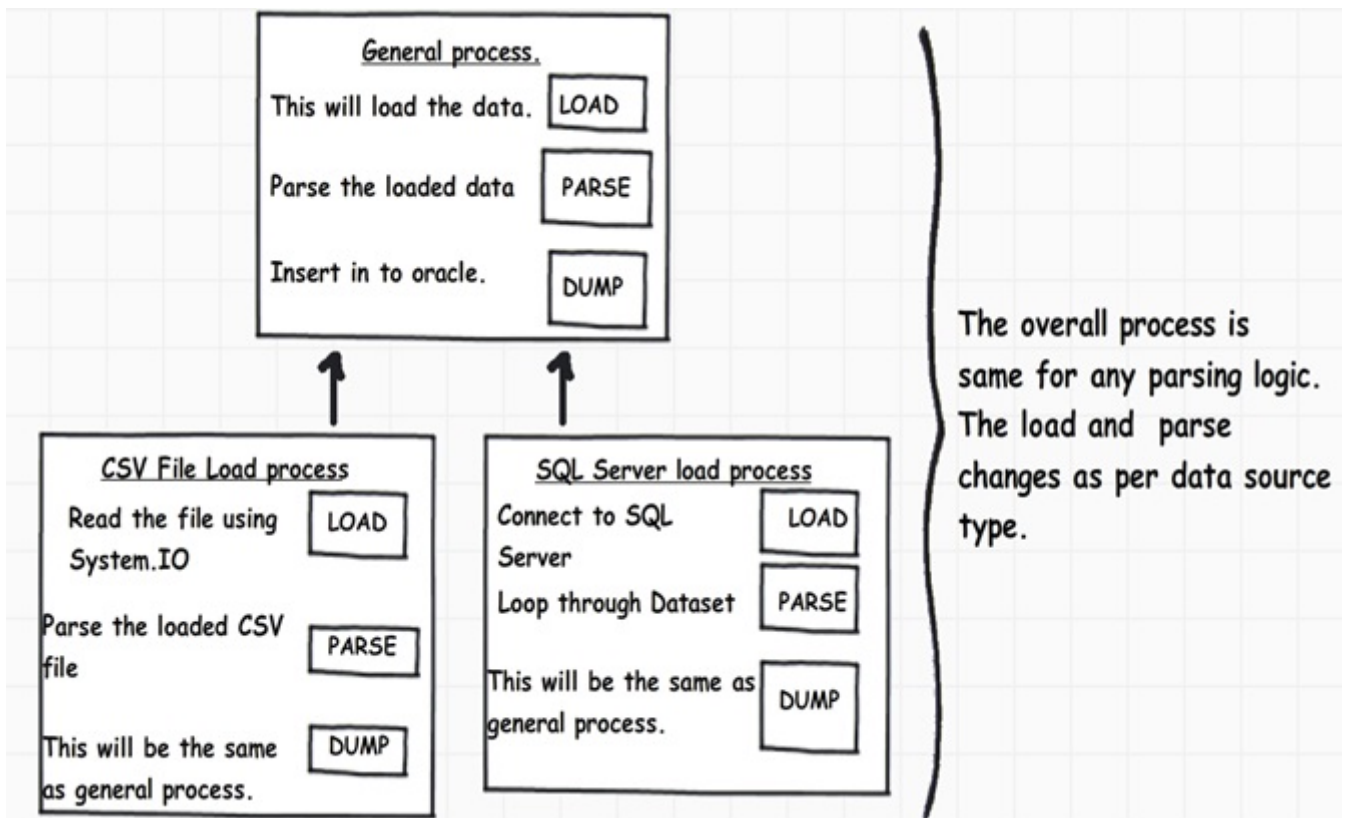
For example below is a simple process to format data and load the same in to oracle. The data can come from various sources like files, SQL server etc. Irrespective from where the data comes, the overall general process is to load the data from the source, parse the data and then dump the same in to oracle.



**Figure: - General Process**

Now we can alter the general process to create a CSV file load process or SQL server load process by overriding 'Load' and 'Parse' sub process implementation.





**Figure: - Template thought Process**

You can see from the above figure how we have altered 'Load' and 'Parse' sub process to generate CSV file and SQL Server load process. The 'Dump' function and the sequence of how the sub processes are called are not altered in the child processes.

In order to implement template pattern we need to follow 4 important steps:-

1. Create the template or the main process by creating a parent abstract class.
2. Create the sub processes by defining abstract methods and functions.
3. Create one method which defines the sequence of how the sub process methods will be called. This method should be defined as a normal method so that we child methods cannot override the same.
4. Finally create the child classes who can go and alter the abstract methods or sub process to define new implementation.

```
public abstract class GeneralParser
{
    protected abstract void Load();

    protected abstract void Parse();
    protected virtual void Dump()
    {
        Console.WriteLine("Dump data in to oracle");
    }
    public void Process()
    {
        Load();
        Parse();
        Dump();
    }
}
```

```
}  
}
```

The 'SqlServerParser' inherits from 'GeneralParser' and overrides the 'Load' and 'Parse' with SQL server implementation.



```
public class SqlServerParser : GeneralParser  
{  
    protected override void Load()  
    {  
        Console.WriteLine("Connect to SQL Server");  
    }  
    protected override void Parse()  
    {  
        Console.WriteLine("Loop through the dataset");  
    }  
}
```

The 'FileParser' inherits from General parser and overrides the 'Load' and 'Parse' methods with file specific implementation.



```
public class FileParser : GeneralParser  
{  
    protected override void Load()  
    {  
        Console.WriteLine("Load the data from the file");  
    }  
    protected override void Parse()  
    {  
        Console.WriteLine("Parse the file data");  
    }  
}
```

From the client you can now call both the parsers.



```
FileParser ObjFileParser = new FileParser();  
ObjFileParser.Process();  
Console.WriteLine("-----");  
SqlServerParser ObjSqlParser = new SqlServerParser();  
ObjSqlParser.Process();  
Console.Read();
```

The outputs of both the parsers are shown below.



```
Load the data from the file  
Parse the file data  
Dump data in to oracle  
-----
```

Connect to SQL Server  
Loop through the dataset  
Dump data [in](#) to oracle