(B) What are design patterns?

Design patterns are documented tried and tested solutions for recurring problems in a given cc and the proposed solution for the same. Design patterns existed in some or other form right frc Let's say if you want to implement a sorting algorithm the first thing comes to mind is bubble bubble sort. Same holds true for design patterns.

(I) 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 Beca
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.
. **Façade**:-A single class that represents an entire subsystem.
. **Flyweight**:-A fine-grained instance used for efficient sharing.
. **Proxy**:-An object representing another object.

Note : To remember structural pattern best is (ABCDFFP)
Behavioral Patterns

. **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.

Note: - Just remember Music..... 2 MICS On TV (MMIICCSSOTV).
Note :- In the further section we will be covering all the above design patterns in a more detail

(A) Can you explain factory pattern?

- Factory pattern is one of the types of creational patterns. You can make out from the nam create something. In software architecture world factory pattern is meant to centralize cre client which has different types of invoices. These invoices are created depending on the ir two issues with the code below:-

- First we have lots of 'new' keyword scattered in the client. In other ways the client is loade can make the client logic very complicated.

  Second issue is that the client needs to be aware of all types of invoices. So if we are addi 'InvoiceWithFooter' we need to reference the new class in the client and recompile the clie

```
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 in to how factory pattern can help us solve the concrete classes 'ClsInvoiceWithHeader' and 'ClsInvoiceWithOutHeader'.

The **first issue** was that these classes are in direct contact with client which leads to lot of 'new' 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. 'ClsInvo This leads to recompiling of the client code when we add new invoice types. For instance if we be changed and recompiled accordingly. To remove this issue we have introduced a commor 'ClsInvoiceWithHeader' and 'ClsInvoiceWithOutHeader' inherit and implement the 'IInvoice' inter

The client references only the 'IInvoice' interface which results in zero connection ('ClsInvoiceWithHeader' and 'ClsInvoiceWithOutHeader'). So now if we add new concrete invoi the client side.

In one line the creation of objects is taken care by 'ClsFactoryInvoice' and the client disconnec 'IInvoice' interface.

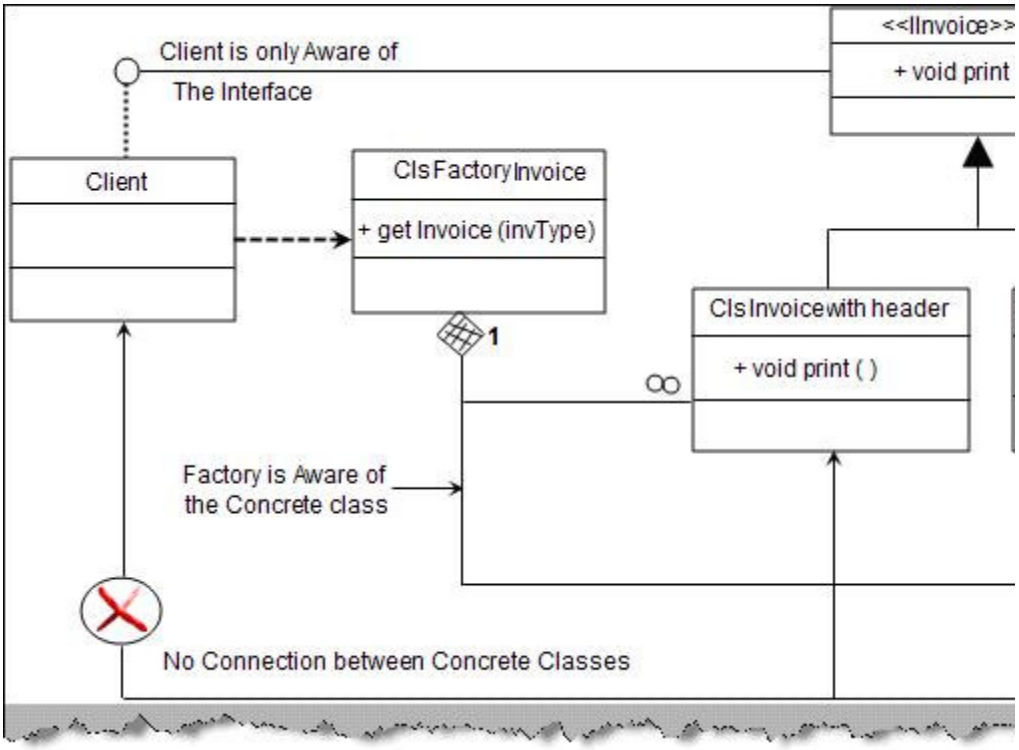

Figure: - Factory pattern

Below are the code snippets of how actually factory pattern can be implemented in C#. In order introduced the invoice interface 'IInvoice'. Both the concrete classes 'ClsInvoiceWithOutHeaders' implement the 'IInvoice' interface.

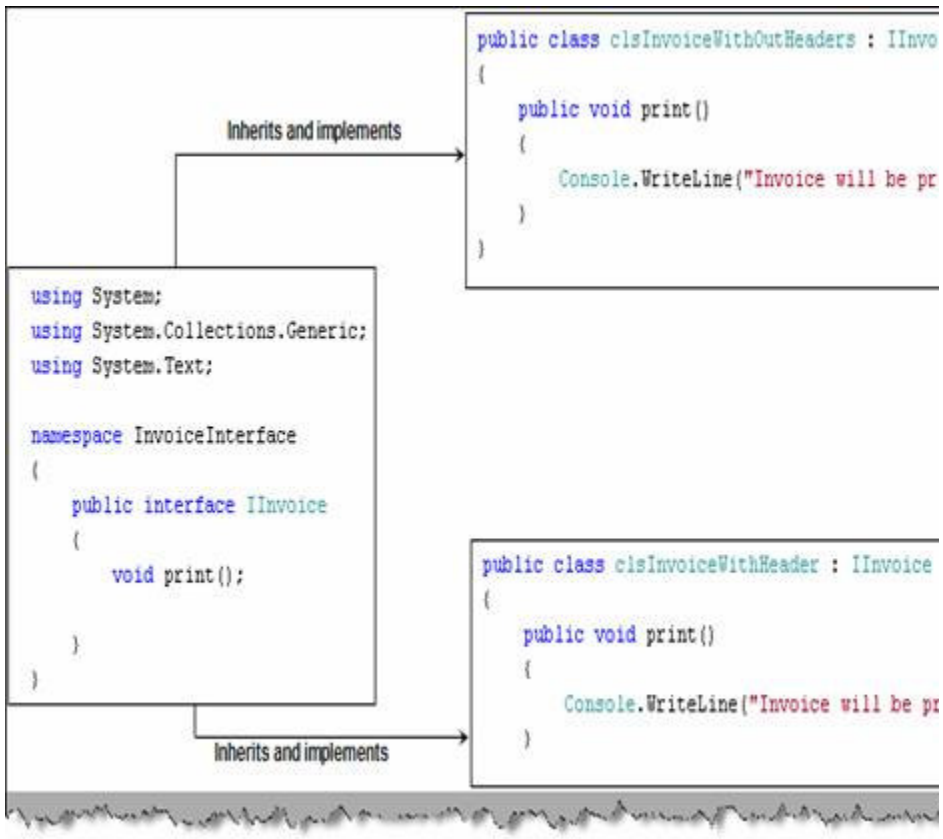

Figure :- Interface and concrete classes

We have also introduced an extra class 'ClsFactoryInvoice' with a function 'getInvoice()' wh depending on 'intInvoiceType' value. In short we have centralized the logic of object creation 'getInvoice' function to generate the invoice classes. One of the most important points to be r and the factory class 'ClsFactoryInvoice' also gives the same type of reference. This helps the c classes, so now when we add new classes and invoice types we do not need to recompile the clie

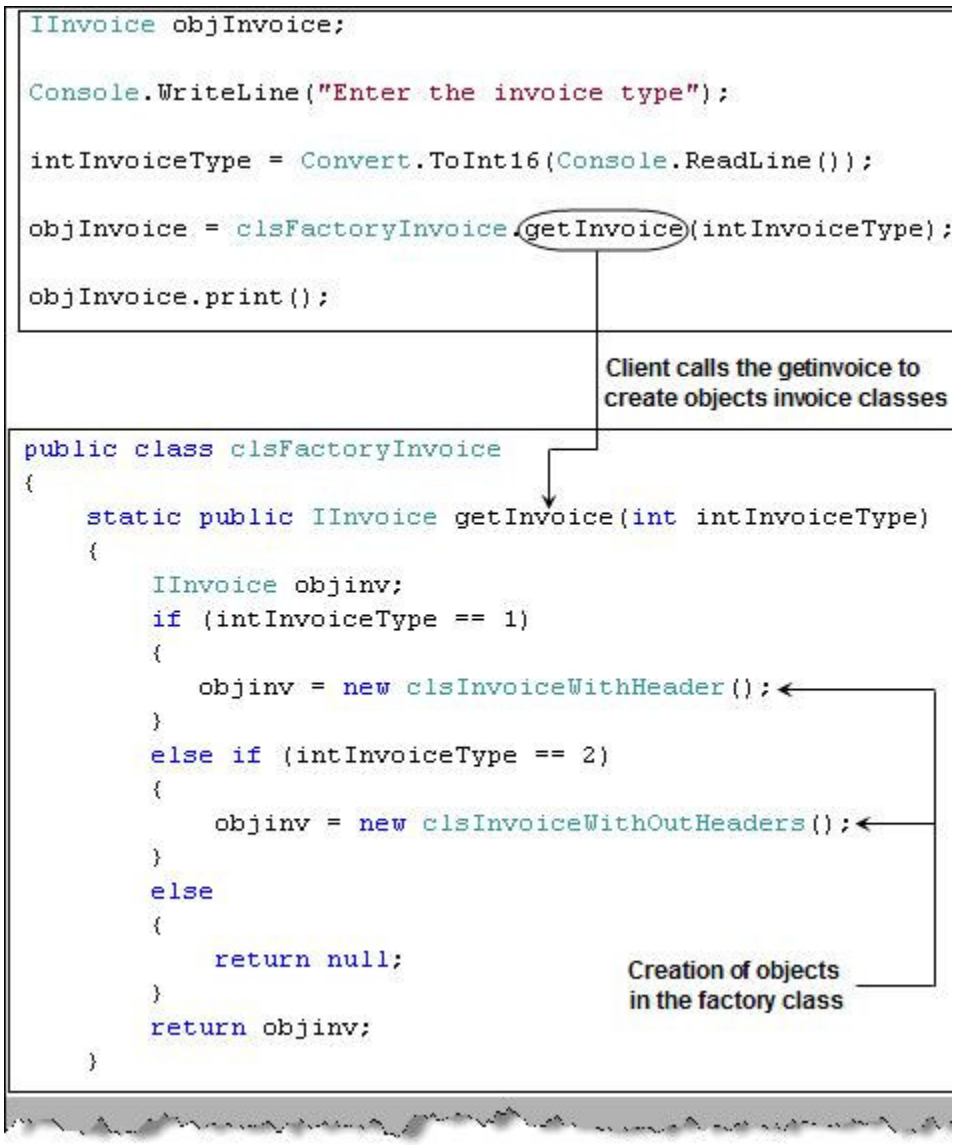

Figure: - Factory class which generates objec

Note :- The above example is given in C# . Even if you are from some other technology you car source code from the CD in 'FactoryPattern' folder.

(I) Can you explain abstract factory pattern?

Abstract factory expands on the basic factory pattern. Abstract factory helps us to unite si interface. So basically all the common factory patterns now inherit from a common abstract fact 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 patterns which leads more simplified interface for the client.

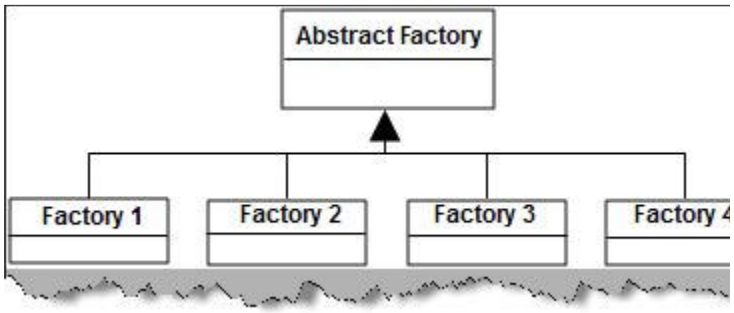

Figure: - Abstract factory unifies related factory p

Now that we know the basic lets try to understand the details of how abstract factory patterns have the factory pattern classes (factory1 and factory2) tied up to a common abstract fact Factory classes stand on the top of concrete classes which are again derived from common int abstract factory' both the concrete classes 'product1' and 'product2' inherits from one interface concrete class will only interact with the abstract factory and the common interface from which t

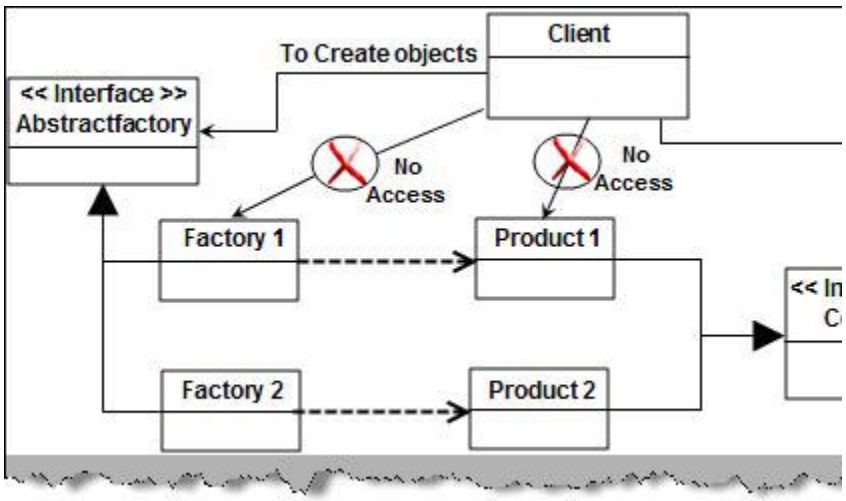

Figure: - Implementation of abstract factor

Now let's have a look at how we can practically implement abstract factory in actual code. activities for textboxes and buttons through their own centralized factory classes 'ClsFactoryB inherit from common interface 'InterfaceRender'. Both the factories 'ClsFactoryButton' and 'Cl 'ClsAbstractFactory'. Figure 'Example for AbstractFactory' shows how these classes are arrange important points to be noted about the client code is that it does not interact with the concrete factory (ClsAbstractFactory) and for calling the concrete class implementation it calls the met 'ClsAbstractFactory' class provides a common interface for both factories 'ClsFactoryButton' and

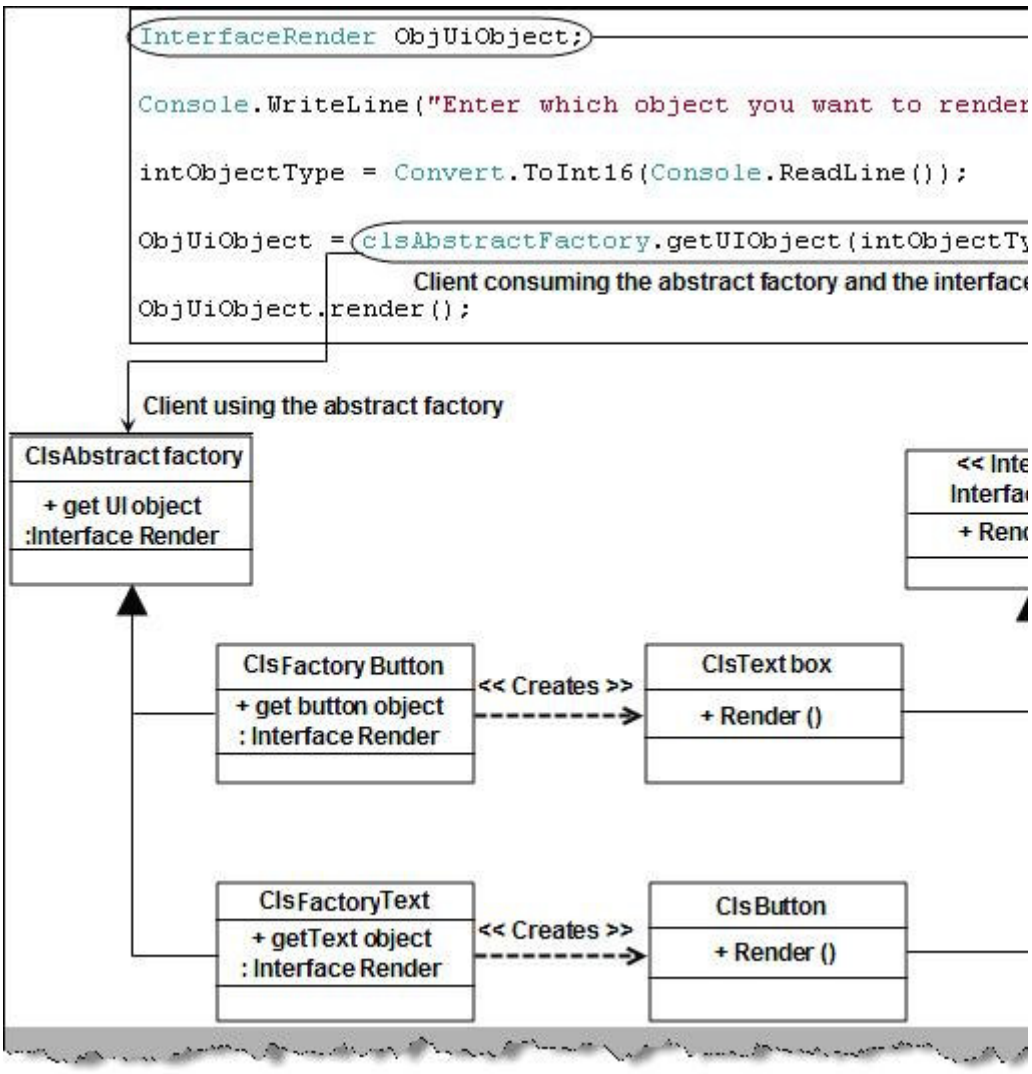

Figure: - Example for abstract factory

Note: - We have provided a code sample in C# in the 'AbstractFactory' folder. People who are f implementation in their own language.

We will just run through the sample code for abstract factory. Below code snippet 'Abstract fa factory pattern classes inherit from abstract factory.

Figure: - Abstract factory and factory code snip

Figure 'Common Interface for concrete classes' how the concrete classes inherits from a commo method 'render' in all the concrete classes.

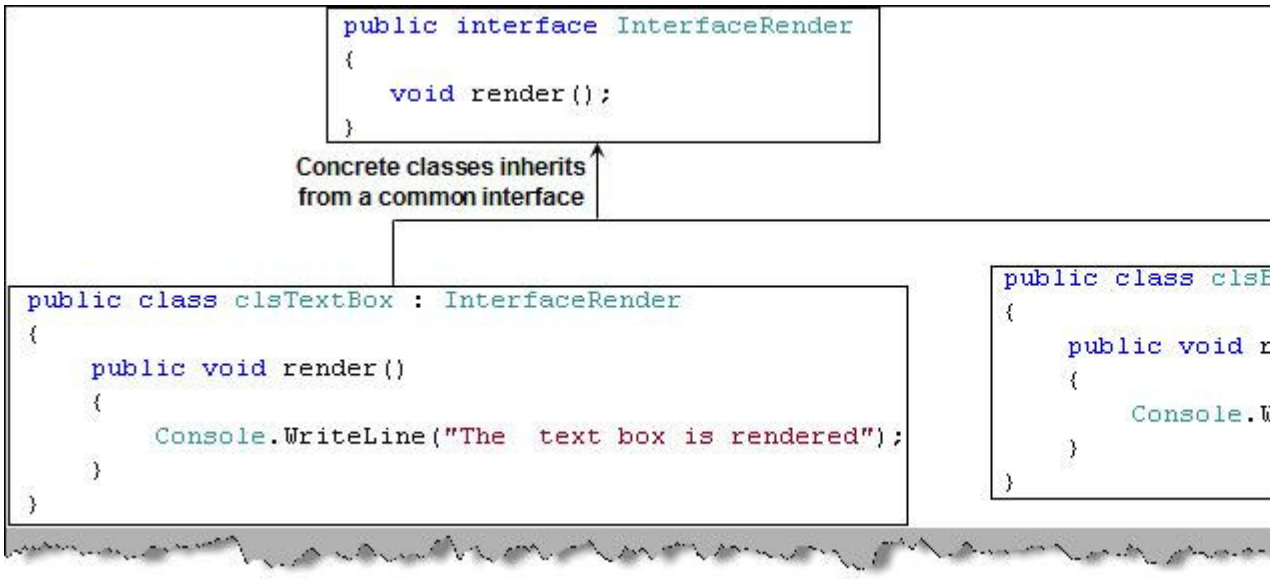

Figure: - Common interface for concrete class

The final thing is the client code which uses the interface 'InterfaceRender' and abstract fac objects. One of the important points about the code is that it is completely isolated from th concrete classes like adding and removing concrete classes does not need client level changes.

```
static void Main(string[] args)
{
    int intObjectType;
    InterfaceRender ObjUiObject;

    Console.WriteLine("Enter which object you want to

    intObjectType = Convert.ToInt16(Console.ReadLine()

    ObjUiObject = clsAbstractFactory.getUIObject(intOb

    ObjUiObject.render();
```


Figure: - Client, interface and abstract factor

(I)Can you explain builder pattern?

Builder falls under the type of creational pattern category. Builder pattern helps us to separate t representation so that the same construction process can create different representations. Build object is very complex. The main objective is to separate the construction of objects and their re construction and representation, we can then get many representations from the same construc

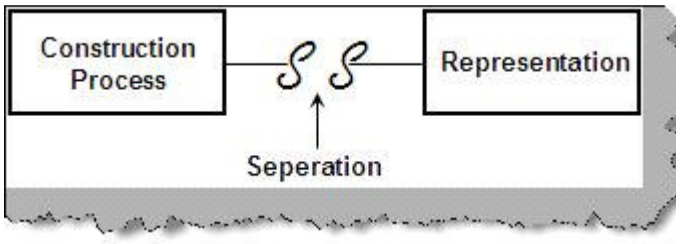

Figure: - Builder concept

To understand what we mean by construction and representation lets take the example of the b

You can see from the figure 'Tea preparation' from the same preparation steps we can get three tea with sugar / milk and tea with out milk).

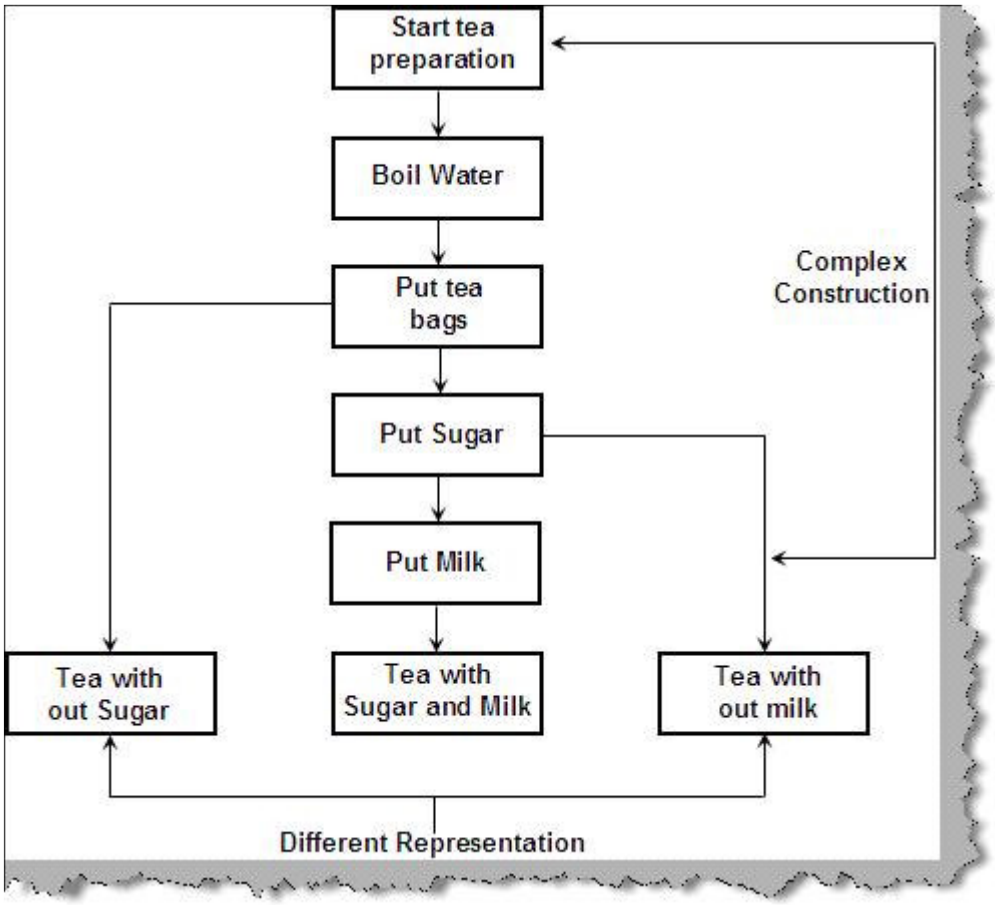

Figure: - Tea preparation

Now let's take a real time example in software world to see how builder can separate the comp have application where we need the same report to be displayed in either 'PDF' or 'EXCEL' form steps to achieve the same. Depending on report type a new report is created, report type is se finally we get the report for display.

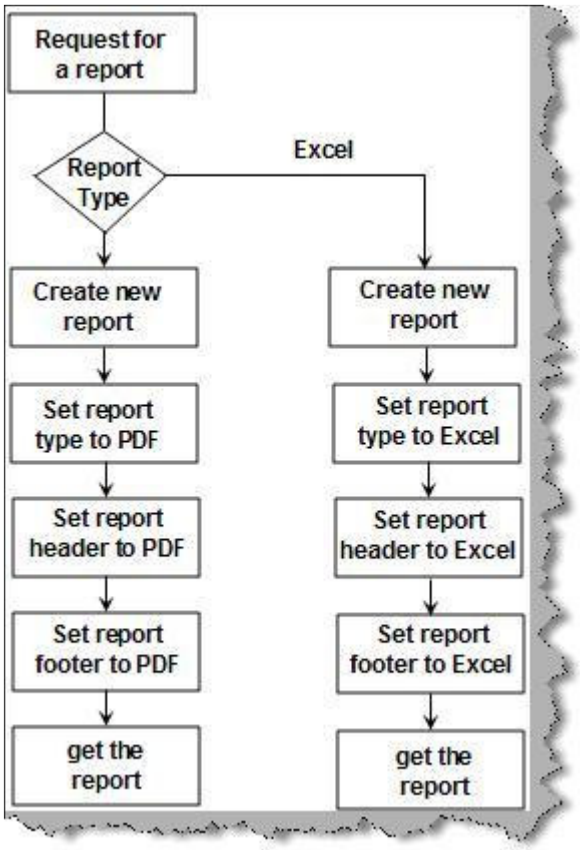

Figure: - Request a report

Now let's take a different view of the problem as shown in figure 'Different View'. The same flow representations and common construction. The construction process is same for both the types representations.

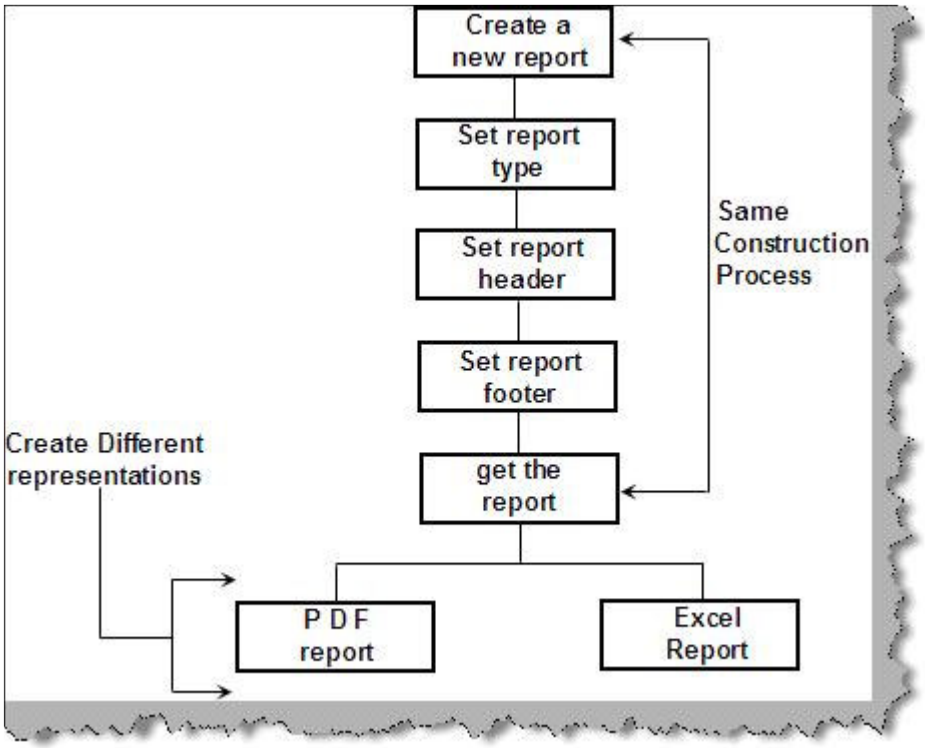

Figure: - Different View

We will take the same report problem and try to solve the same using builder patterns. There ar
builder patterns.

. **Builder**: - Builder is responsible for defining the construction process for individual parts. Builc
and configure the product.
. **Director**: - Director takes those individual processes from the builder and defines the sequenc
. **Product**: - Product is the final object which is produced from the builder and director coordinat

First let's have a look at the builder class hierarchy. We have a abstract class called as 'ReportBu
'ReportPDF' builder and 'ReportEXCEL' builder will be built.

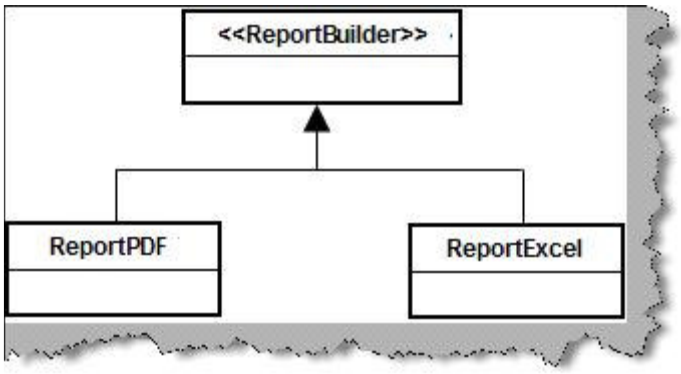

Figure: - Builder class hierarchy

Figure 'Builder classes in actual code' shows the methods of the classes. To generate report we r
type (to EXCEL or PDF) , set report headers , set the report footers and finally get the report. W
'PDF' (ReportPDF) and other for 'EXCEL' (ReportExcel). These two custom builders define there c

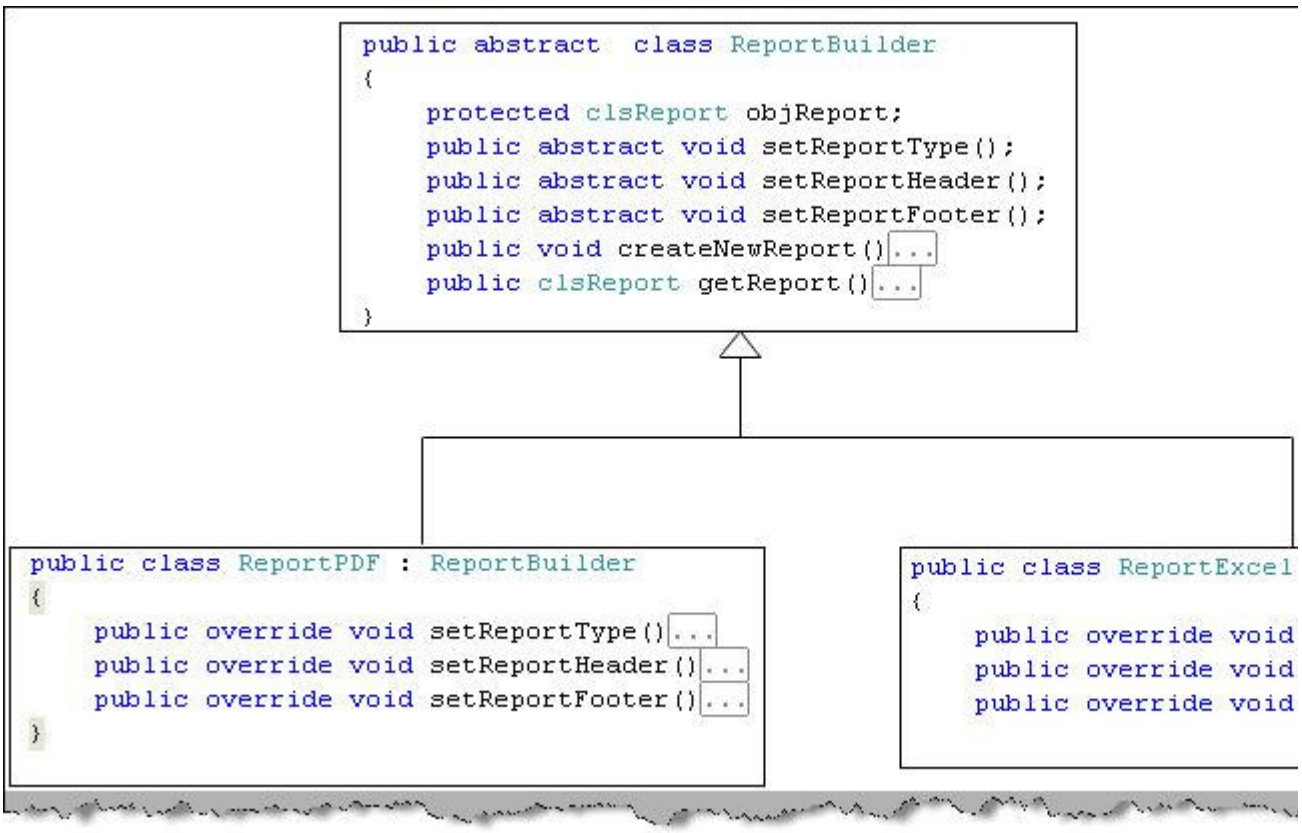

Figure: - Builder classes in actual code

Now let's understand how director will work. Class 'clsDirector' takes the builder and calls the inc
So director is like a driver who takes all the individual processes and calls them in sequential ma
report in this case. Figure 'Director in action' shows how the method 'MakeReport' calls the indiv
PDF or EXCEL.

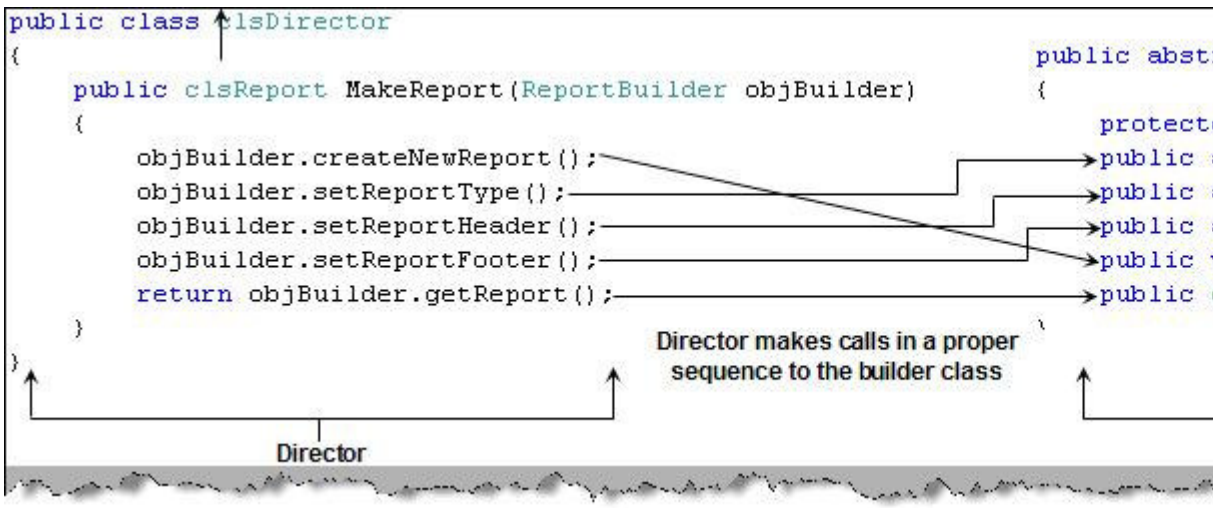

Figure: - Director in action

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

```
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' show
Client creates the object of the director class and passes the appropriate builder to initialize the
is initialized/created and finally sent to the client.

```
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
{
    public override vo
    {
        objReport.strR
    }
    public override vo
    {
        objReport.setR
    }
    public override vo
    {
        objReport.setR
    }
}
```

PDF b

```
public class ReportExcel
{
    public override void
    {
        objReport.strRep
    }
    public override void
    {
        objReport.setRep
    }
    public override void
    {
        objReport.setRep
    }
```

Figure: - Client, builder, director and produc

The output is something like this. We can see two report types displayed with their headers accc

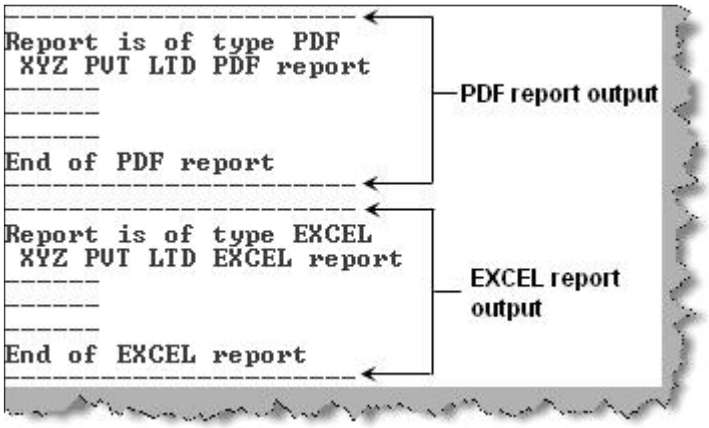

Figure: - Final output of builder

Note :- In CD we have provided the above code in C# in 'BuilderPattern' folder.

(I) Can you explain prototype pattern?

Prototype pattern falls in the section of creational pattern. It gives us a way to create new objec one sentence we clone the existing object with its data. By cloning any changes to the cloned ob you are thinking by just setting objects we can get a clone then you have mistaken it. By setting of object BYREF. So changing the new object also changed the original object. To understand 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 val

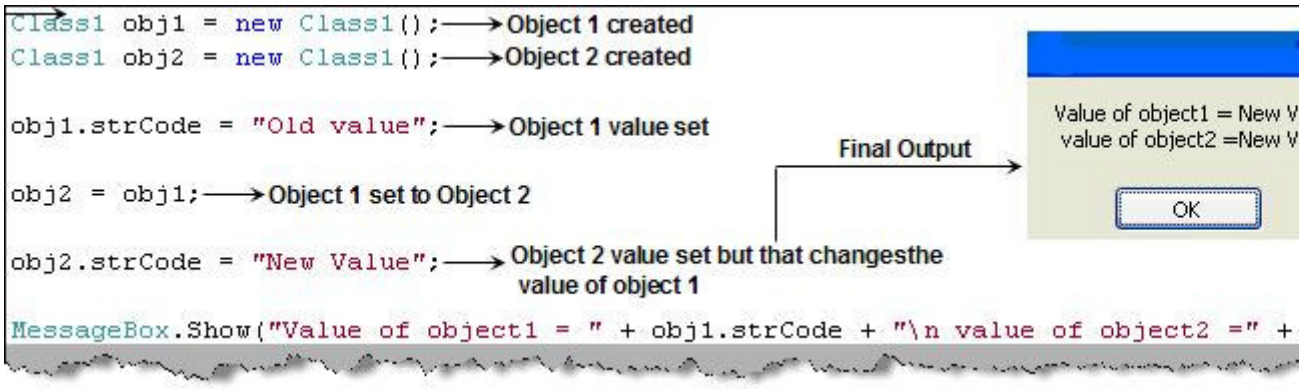

Figure :- BYREf

The conclusion of the above example is that objects when set to other objects are set BYREF. S old object value.

There are many instances when we want the new copy object changes should not affect the old

Lets look how we can achieve the same using C#. In the below figure 'Prototype in action' w needs to be cloned. This can be achieved in C# my using the 'MemberWiseClone' method. In J same. In the same code we have also shown the client code. We have created two objects of th to 'obj2' will not affect 'obj1' as it's a complete cloned copy.

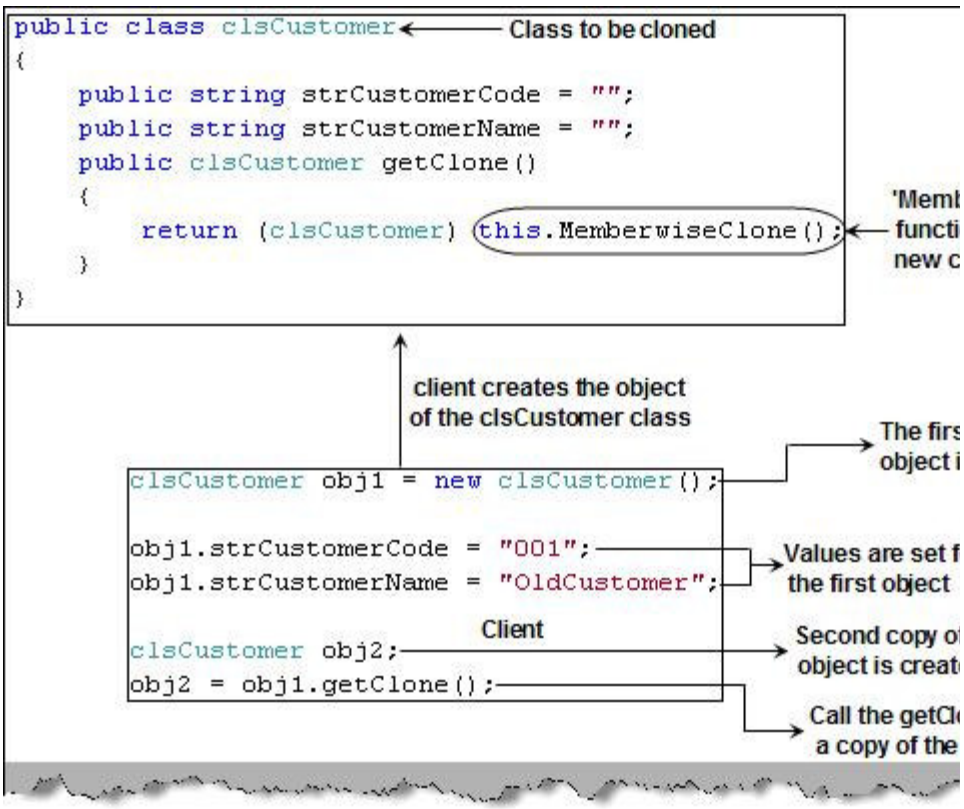

Figure: - Prototype in action

Note :- You can get the above sample in the CD in 'Prototype' folder. In C# we use the 'Membe 'Clone' function to achieve the same.

(A) 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 only that object is cloned, any objects containing in that object is not cloned. For instance consic customer class and we have an address class aggregated inside the customer class. 'MemberWis 'ClsCustomer' but not the 'ClsAddress' class. So we added the 'MemberWiseClone' function in the 'getClone' function we call the parent cloning function and also the child cloning function, which the parent objects are cloned with their containing objects it's called as deep cloning and when c cloning.

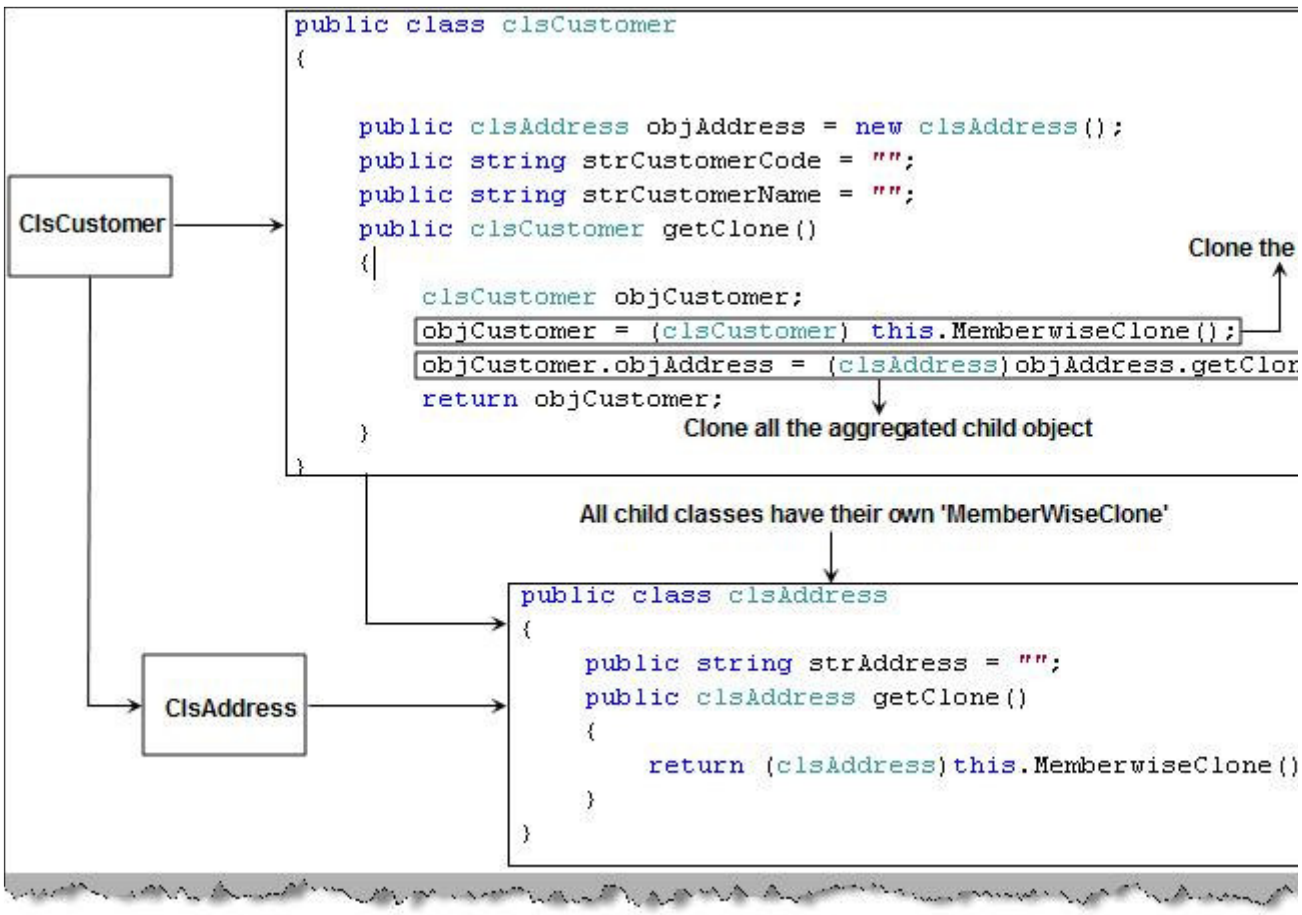

Figure: - Deep cloning in action

(B) Can you explain singleton pattern?

There are situations in a project where we want only one instance of the object to be created create an instance of the object from outside. There is only one instance of the class which is s make a singleton pattern:-

1) Define the constructor as private.
2) Define the instances and methods as static.

Below is a code snippet of a singleton in C#. We have defined the constructor as private, defin keyword as shown in the below code snippet figure 'Singleton in action'. The static keyword ens created and you can all the methods of the class with out creating the object. As we have ma class directly.

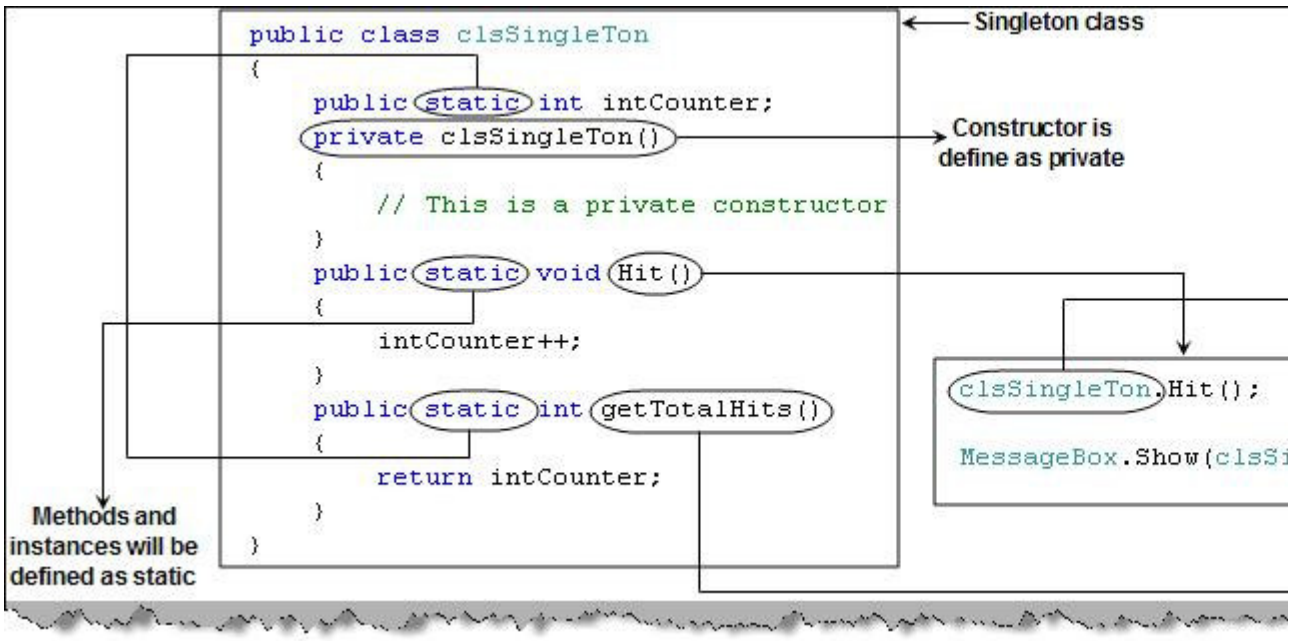

Figure: - Singleton in action

Note :- In JAVA to create singleton classes we use the STATIC keyword , so its same as in C#. Y the 'singleton' folder.

(A) Can you explain command patterns?

Command pattern allows a request to exist as an object. Ok let's understand what it means. Cor different actions depending on which menu is clicked. So depending on which menu is clicked we text in the action string. Depending on the action string we will execute the action. The bad thin which makes the coding more cryptic.

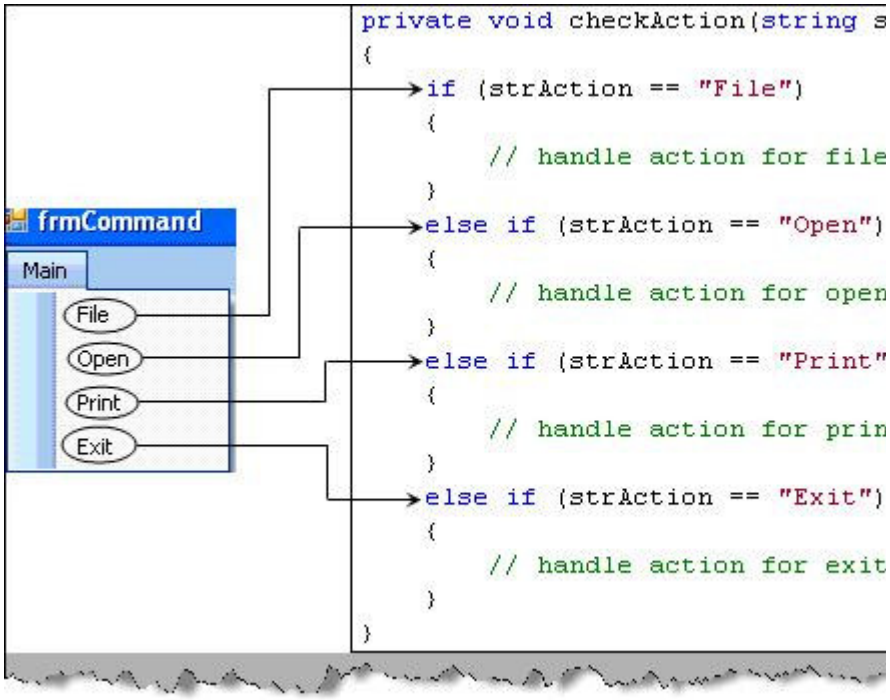

Figure: - Menu and Commands

Command pattern moves the above action in to objects. These objects when executed actually e
As said previously every command is an object. We first prepare individual classes for every ac
actions are wrapped in to classes like Exit action is wrapped in 'clsExecuteExit' , open action
wrapped in 'clsExecutePrint' and so on. All these classes are inherited from a common interface '

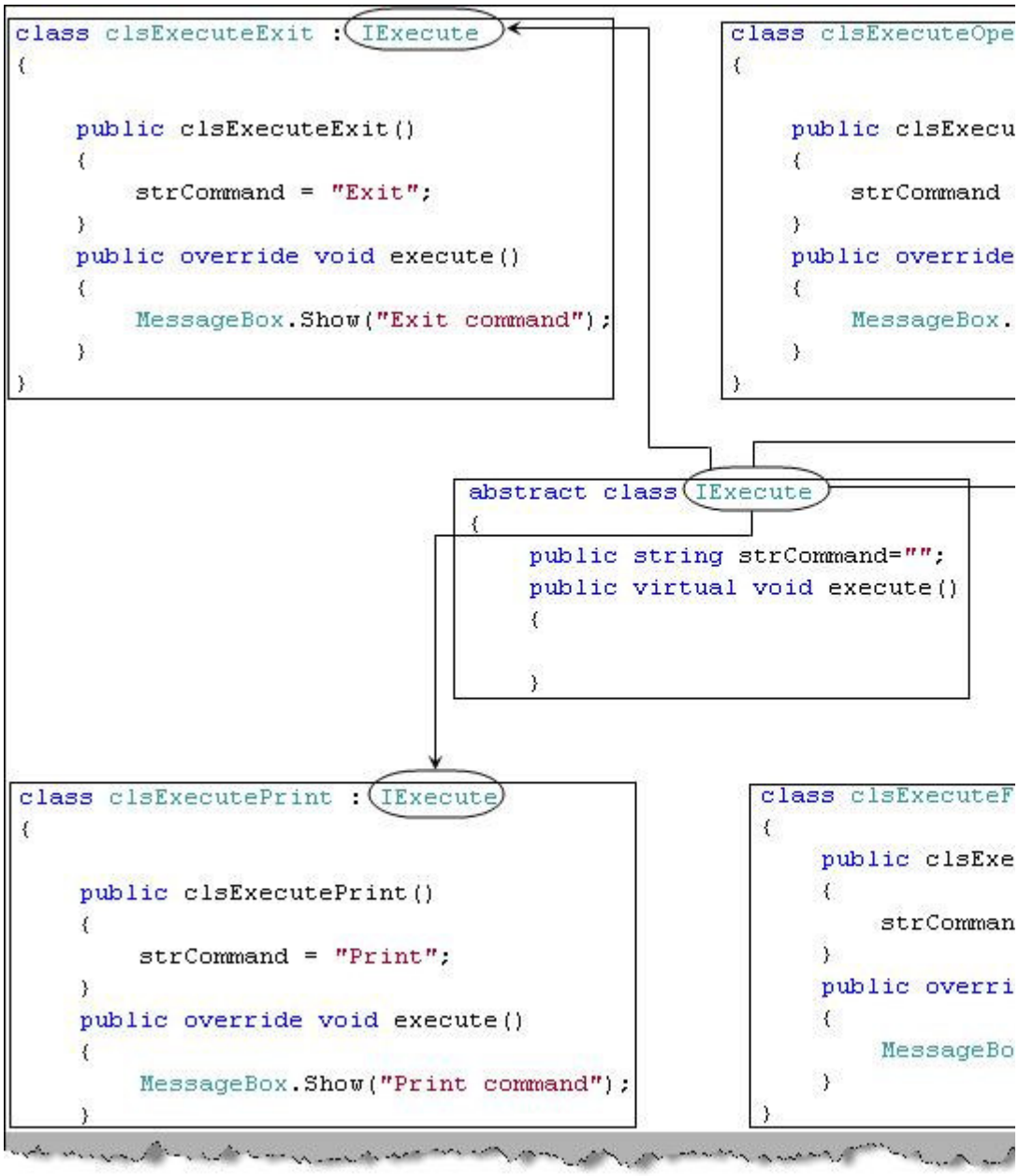

Figure: - Objects and Command

Using all the action classes we can now make the invoker. The main work of invoker is to map th
So we have added all the actions in one collection i.e. the arraylist. We have exposed a method
back the abstract object 'IExecute'. The client code is now neat and clean. All the 'IF' conditions

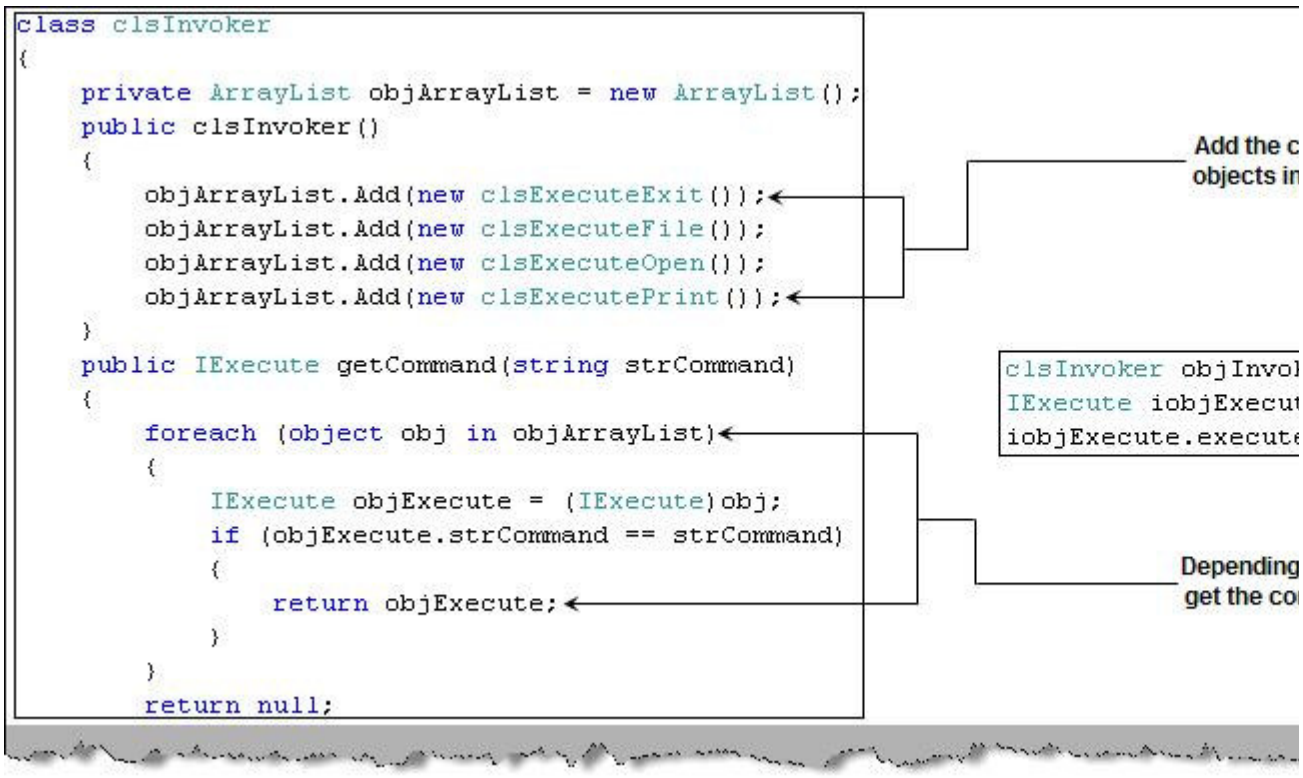

Figure: - Invoker and the clean client