**Section16 Command Design Pattern**

**Notes: -**

**1-Command Design Pattern: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.**

**In the below example we create class calcoulator which provide the business function and using it across abstract and sub class inherit from it and call them iniside it**

**And create Invoker class which initialize the Calculator and Concrete class and**

**Example: -**

**namespace DoFactoryCommandPatternPro.Commands{**

**//Command use Reciever as parameter and provide abstract method as below**

**abstract class Command{**

**protected Receiver receiver;**

**public Command(Receiver receiver){this.receiver = receiver;}**

**public abstract void Execute();}}**

**using System;**

**namespace DoFactoryCommandPatternPro.Commands{**

**//this class represnt the command that we want to operate**

**//1- on the Command abstract class**

**//2- on the ConcreteCommand child class that inherit from Command abstract class**

**class Receiver{public void Action(){Console.WriteLine("Called Receiver.Action()");}}}**

**namespace DoFactoryCommandPatternPro.Commands{**

**//this class is what we called on the Program.cs and using reciever as parameter to assign**

**class ConcreteCommand : Command{**

**public ConcreteCommand(Receiver receiver) :base(receiver){}**

**public override void Execute(){receiver.Action();}}}**

**namespace DoFactoryCommandPatternPro.Commands{**

**//the invoker will call the ConcreteCommand that contains the Receiver class as below**

**class Invoker{**

**private Command \_command;**

**public void SetCommand(Command command){this.\_command = command;}**

**public void ExecuteCommand(){\_command.Execute();}}}**

**using DoFactoryCommandPatternPro.Commands;**

**using System;**

**namespace DoFactoryCommandPatternPro{**

**class Program{**

**static void Main(string[] args){**

**// Create receiver, assign to ConcreteCommand , Create Invoker**

**Receiver receiver = new Receiver();**

**Command command = new ConcreteCommand(receiver);**

**Invoker invoker = new Invoker();**

**// Set and execute command on the Invoker instance**

**invoker.SetCommand(command);**

**invoker.ExecuteCommand();**

**Console.ReadKey();}}}**

**Example: -**

**This example provide the undo and redo operation support as below**

**namespace DoFactoryCommandPattern2Pro{**

**//we define the interface Command**

**abstract class Command{**

**public abstract void Execute();**

**public abstract void UnExecute();}}**

**//the class Calcoulator will contains the functionality and using it inside the ConcreteCalcoulator**

**using System;**

**namespace DoFactoryCommandPattern2Pro{**

**class Calculator{**

**private int \_curr = 0;**

**public void Operation(char @operator, int operand){**

**switch (@operator){**

**case '+': \_curr += operand; break;**

**case '-': \_curr -= operand; break;**

**case '\*': \_curr \*= operand; break;**

**case '/': \_curr /= operand; break;}**

**Console.WriteLine("Current value = {0,3} (following {1} {2})",\_curr, @operator, operand);}}}**

**using System;**

**namespace DoFactoryCommandPattern2Pro{**

**//the child class inherit from the interface and pass the Calcoulator as parameter**

**class CalculatorCommand : Command{**

**private char \_operator;**

**private int \_operand;**

**private Calculator \_calculator;**

**public CalculatorCommand(Calculator calculator,char @operator, int operand){**

**this.\_calculator = calculator;**

**this.\_operator = @operator;**

**this.\_operand = operand;}**

**// Gets operator**

**public char Operator{set { \_operator = value; }}**

**// Get operand**

**public int Operand{set { \_operand = value; }}**

**// Execute new command**

**public override void Execute(){\_calculator.Operation(\_operator, \_operand);}**

**// Unexecute last command**

**public override void UnExecute(){\_calculator.Operation(Undo(\_operator), \_operand);}**

**// Returns opposite operator for given operator**

**private char Undo(char @operator){**

**switch (@operator){**

**case '+': return '-';**

**case '-': return '+';**

**case '\*': return '/';**

**case '/': return '\*';**

**default:**

**throw new**

**ArgumentException("@operator");}}}}**

**using System;**

**using System.Collections.Generic;**

**using System.Text;**

**namespace DoFactoryCommandPattern2Pro{**

**//on the Invoker we initiliaze all the Calcoulator which is Command and the CalcoulatorCommand**

**class User{**

**// Initializers**

**private Calculator \_calculator = new Calculator();**

**private List<Command> \_commands = new List<Command>();**

**private int \_current = 0;**

**public void Redo(int levels){**

**Console.WriteLine("\n---- Redo {0} levels ", levels);**

**// Perform redo operations**

**for (int i = 0; i < levels; i++){**

**if (\_current < \_commands.Count - 1){**

**Command command = \_commands[\_current++];**

**command.Execute();}}}**

**public void Undo(int levels){**

**Console.WriteLine("\n---- Undo {0} levels ", levels);**

**// Perform undo operations**

**for (int i = 0; i < levels; i++){**

**if (\_current > 0){**

**Command command = \_commands[--\_current] as Command;**

**command.UnExecute();}}}**

**public void Compute(char @operator, int operand){**

**// Create command operation and execute it**

**Command command = new CalculatorCommand(**

**\_calculator, @operator, operand);**

**command.Execute();**

**// Add command to undo list**

**\_commands.Add(command);**

**\_current++;}}}**

**using System;**

**namespace DoFactoryCommandPattern2Pro{**

**class Program{**

**static void Main(string[] args){**

**// Create user and let her compute**

**User user = new User();**

**// User presses calculator buttons**

**user.Compute('+', 100);**

**user.Compute('-', 50);**

**user.Compute('\*', 10);**

**user.Compute('/', 2);**

**// Undo 4 commands**

**user.Undo(4);**

**// Redo 3 commands**

**user.Redo(3);**

**Console.ReadKey();}}}**