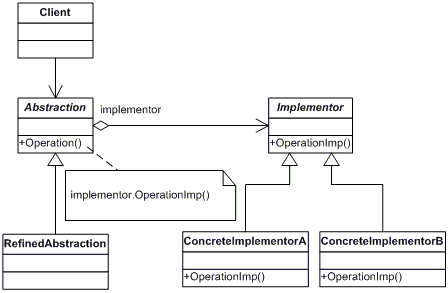
**Section08 Bridge Design Pattern**

**Notes: -**

**Decouple an abstraction from its implementation so that the two can vary independently**.

**By make all classes inherit from the same interface and using it as property on the Abstraction class and with this we can generate what we want of the target termail classes**



Example:-

**using static System.Console;**

**namespace doFactoryBridge{**

**class ConcreteImplementorA : Implementor{**

**public override void Operation()**

**{WriteLine("ConcreteImplementorA Operation");}}}**

**using static System.Console;**

**namespace doFactoryBridge{**

**class ConcreteImplementorB : Implementor{**

**public override void Operation(){**

**WriteLine("ConcreteImplementorB Operation");}}}**

**namespace doFactoryBridge{**

**abstract class Implementor{**

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

**//we inject the Implementor as property on the Abstraction class to access to any class that inherit //from the Implementor**

**namespace doFactoryBridge{**

**class Abstraction{**

**protected Implementor implementor;**

**public Implementor Implementor{set { implementor = value; }}**

**public virtual void Operation(){implementor.Operation();}}}**

**namespace doFactoryBridge{**

**class RefinedAbstraction : Abstraction{**

**public override void Operation(){implementor.Operation();}}}**

**using System;**

**namespace doFactoryBridge{**

**//Summary: Decouple an abstraction from its implementation so that the two can vary independently.**

**//so all the terminal classes that inherit from the Implementor like ConcreteImplementorA,ConcreteImplementorB**

**//the base class are used as property on the Abstraction class and use the property to generate what we want**

**class Program{**

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

**Abstraction ab = new RefinedAbstraction();**

**// Set implementation and call**

**ab.Implementor = new ConcreteImplementorA();**

**ab.Operation();**

**// Change implemention and call**

**ab.Implementor = new ConcreteImplementorB();**

**ab.Operation();**

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

Example:-

**using System;**

**namespace doFactoryBridgeSecond{**

**//Abstraction class hold the DataObject which is abstract / interface that all sub class inherit from abstract class CustomersBase{**

**private DataObject \_dataObject;**

**protected string group;**

**public CustomersBase(string group){this.group = group;}**

**public DataObject Data{set { \_dataObject = value; }get { return \_dataObject; }}**

**public virtual void Next(){\_dataObject.NextRecord();}**

**public virtual void Prior(){\_dataObject.PriorRecord();}**

**public virtual void Add(string customer){\_dataObject.AddRecord(customer);}**

**public virtual void Delete(string customer){\_dataObject.DeleteRecord(customer);}**

**public virtual void Show(){\_dataObject.ShowRecord();}**

**public virtual void ShowAll(){**

**Console.WriteLine("Customer Group: " + group);**

**\_dataObject.ShowAllRecords();}}}**

**using System;**

**namespace doFactoryBridgeSecond{**

**//terminal class that inherit from the CustomerBase abstract class**

**class Customers : CustomersBase{**

**// Constructor**

**public Customers(string group): base(group){}**

**public override void ShowAll(){**

**// Add separator lines**

**Console.WriteLine();**

**Console.WriteLine("------------------------");**

**base.ShowAll();**

**Console.WriteLine("------------------------");}}}**

**namespace doFactoryBridgeSecond{**

**abstract class DataObject{**

**public abstract void NextRecord();**

**public abstract void PriorRecord();**

**public abstract void AddRecord(string name);**

**public abstract void DeleteRecord(string name);**

**public abstract void ShowRecord();**

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

**using System;**

**using System.Collections.Generic;**

**namespace doFactoryBridgeSecond{**

**class CustomersData : DataObject{**

**private List<string> \_customers = new List<string>();**

**private int \_current = 0;**

**public CustomersData(){**

**// Loaded from a database**

**\_customers.Add("Jim Jones");**

**\_customers.Add("Samual Jackson");**

**\_customers.Add("Allen Good");**

**\_customers.Add("Ann Stills");**

**\_customers.Add("Lisa Giolani");}**

**public override void NextRecord(){**

**if (\_current <= \_customers.Count - 1){\_current++;}}**

**public override void PriorRecord(){**

**if (\_current > 0){\_current--;}}**

**public override void AddRecord(string customer){\_customers.Add(customer);}**

**public override void DeleteRecord(string customer){\_customers.Remove(customer);}**

**public override void ShowRecord(){Console.WriteLine(\_customers[\_current]);}**

**public override void ShowAllRecords(){**

**foreach (string customer in \_customers){Console.WriteLine(" " + customer);}}}}**

**using System;**

**namespace doFactoryBridgeSecond{**

**class Program{**

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

**// Create RefinedAbstraction**

**Customers customers = new Customers("Chicago");**

**// Set ConcreteImplementor**

**customers.Data = new CustomersData();**

**// Exercise the bridge**

**customers.Show();**

**customers.Next();**

**customers.Show();**

**customers.Next();**

**customers.Show();**

**customers.Add("Henry Velasquez");**

**customers.ShowAll();**

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

**Example:-**

**In the below example we apply Bridge design pattern by make all terminal class inherit from IRenderer and using it as property on the abstract class that all class like circle inherit from Shape class to access to its properiteis**

**//IRender interface that represent the abstraction of all terminal classes as below**

**namespace BridgePatternPro.Interface{**

**public interface IRenderer{**

**void RenderCircle(float radius);}}**

**using BridgePatternPro.Interface;**

**using static System.Console;**

**namespace BridgePatternPro.Models{**

**public class VectorRenderer : IRenderer{**

**public void RenderCircle(float radius){WriteLine($"Drawing a circle of radius {radius}");}}}**

**using BridgePatternPro.Interface;**

**using static System.Console;**

**namespace BridgePatternPro.Models{**

**public class RasterRenderer : IRenderer{**

**public void RenderCircle(float radius){**

**WriteLine($"Drawing pixels for circle of radius {radius}");}}}**

**//we using IRender as proerpty on the Shape abstract class and its allow the child class like Circle //to use this property to access to target terminal class we want**

**using BridgePatternPro.Interface;**

**namespace BridgePatternPro.Models{**

**public abstract class Shape{**

**protected IRenderer renderer;**

**// a bridge between the shape that's being drawn an**

**// the component which actually draws it**

**public Shape(IRenderer renderer){this.renderer = renderer;}**

**public abstract void Draw();**

**public abstract void Resize(float factor);}}**

**//this is the abstract class that use IRenderer as property and make child classes like Circle inherit //from it as below**

**using BridgePatternPro.Interface;**

**namespace BridgePatternPro.Models{**

**public class Circle : Shape{**

**private float radius;**

**//we can access to the instance by create anther instance of IRenderer of the terminal class**

**public Circle(IRenderer renderer, float radius) : base(renderer){this.radius = radius;}**

**public override void Draw(){renderer.RenderCircle(radius);}**

**public override void Resize(float factor){radius \*= factor;}}}**

**using Autofac;**

**using BridgePatternPro.Interface;**

**using BridgePatternPro.Models;**

**using static System.Console;**

**namespace BridgePatternPro{**

**class Program{**

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

**var raster = new RasterRenderer();**

**var vector = new VectorRenderer();**

**var circle = new Circle(vector, 5);**

**circle.Draw();**

**circle.Resize(2);**

**circle.Draw();**

**//by using Autofac we can apply the Bridge design pattern by provide custom register to any class //that inherit from IRenderer plus any parameter of type float**

**var cb = new ContainerBuilder();**

**cb.RegisterType<VectorRenderer>().As<IRenderer>();**

**cb.Register((c, p) => new Circle(c.Resolve<IRenderer>(),**

**p.Positional<float>(0)));**

**using (var c = cb.Build()){**

**var circle2 = c.Resolve<Circle>(**

**new PositionalParameter(0, 5.0f));**

**circle2.Draw();**

**circle2.Resize(2);**

**circle2.Draw();}}}}**