**Section07 Adapter Design Pattern**

**Lesson01 What is the Adapter Design Pattern**

**Notes: -**

**1-this is the first pattern for the Structural pattern**

**2-Adapter design pattern is talk about getting the interface you want from the interface you have**

**(A construct which adapt an existing interface X to conform to the required interface Y)**

**Example: -**

**1-some systems require interface A and other systems require interface B and you want to interact between these systems, so you need to use Adapter design pattern**

**Lesson02 Vector / Raster Demo**

**Notes: -**

**1-the bad side of the Adapter design pattern is that it will generate a lot of temporary information**

**2-in this example adapter is just class inherit from Collection<Point> which is used to convert list of lines that represent shape into list of points**

**Example:-**

**using System.Collections.ObjectModel;**

**namespace AdapterSimplePro{**

**//the base class of multiple shapes**

**public class VectorObject : Collection<Line>{}}**

**namespace AdapterSimplePro{**

**public class VectorRectangle : VectorObject{**

**public VectorRectangle(int x, int y, int width, int height){**

**Add(new Line(new Point(x, y), new Point(x + width, y)));**

**Add(new Line(new Point(x+width, y), new Point(x + width, y+height)));**

**Add(new Line(new Point(x, y), new Point(x , y + height)));**

**Add(new Line(new Point(x, y), new Point(x + width, y + height)));}}}**

**namespace AdapterSimplePro{**

**public class Point{**

**public int X,Y;**

**public Point(int x, int y){X = x;Y = y;}**

**public override string ToString(){**

**return $"{nameof(X)}: {X}, {nameof(Y)}: {Y}";}}}**

**namespace AdapterSimplePro{**

**//Line contains two proeprty Point of Start , End**

**public class Line{**

**public Point Start;**

**public Point End;**

**public Line(Point start, Point end){**

**this.Start = start;**

**this.End = end;}}}**

**using System;**

**using System.Collections.ObjectModel;**

**using static System.Console;**

**namespace AdapterSimplePro{**

**//we using LineToPointAdapter as adapter to convert line into collection of points**

**public class LineToPointAdapter : Collection<Point>{**

**private static int count = 0;**

**public LineToPointAdapter(Line line){**

**WriteLine($"{++count}: Generating points for line [{line.Start.X},{line.Start.Y}]-[{line.End.X},{line.End.Y}] (no caching)");**

**int left = Math.Min(line.Start.X, line.End.X);**

**int right = Math.Max(line.Start.X, line.End.X);**

**int top = Math.Min(line.Start.Y, line.End.Y);**

**int bottom = Math.Max(line.Start.Y, line.End.Y);**

**int dx = right - left;**

**int dy = line.End.Y - line.Start.Y;**

**if (dx == 0){**

**for (int y = top; y <= bottom; ++y){Add(new Point(left, y));}}**

**else if (dy == 0){**

**for (int x = left; x <= right; ++x){Add(new Point(x, top));}}}}}**

**on the main entry point**

**using MoreLinq;**

**using System.Collections.Generic;**

**using static System.Console;**

**namespace AdapterSimplePro{**

**//we will use adapter design pattern which convert the vector object which is rectangle**

**//into list of lines that will converted into collection of points**

**//we have VectorObject which inherit from Collection<Line>**

**//we have VectorRectangle which inherit from VectorObject**

**//we have Line > Point Start,Point End**

**class Program{**

**public static readonly List<VectorObject> vectorObjects**

**= new List<VectorObject>(){**

**new VectorRectangle(1,1,10,10),**

**new VectorRectangle(3,3,6,6)};**

**public static void DrawPoint(Point p){Write(".");}**

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

**Draw();**

**Draw();**

**ReadLine();**

**WriteLine("Hello World!");}**

**private static void Draw(){**

**foreach (var vo in vectorObjects){**

**foreach (var line in vo){**

**var adapter = new LineToPointAdapter(line);**

**adapter.ForEach(pt => {WriteLine($"({pt.X},{pt.Y})");});}}**

**ReadLine();}}}**

**Lesson03 Caching Adapter**

**Notes:-**

**1-one of the bad sides of the Adapter design pattern is that it generate a lot of temp data**

**2-so we can use caching adapter pattern to store temp without replicate them on each rectangle shape by check on Equal override method**

**Example:-**

**namespace AdapterCachingPro{**

**public class Point{**

**public int X;**

**public int Y;**

**public Point(int x, int y){**

**this.X = x;**

**this.Y = y;}**

**protected bool Equals(Point other)**

**{return X == other.X && Y == other.Y;}**

**//it will resolve the issue of the replicate temp data**

**//make check equal by reference if it equals return true**

**//and then check for the two X , Y**

**public override bool Equals(object obj){**

**if (ReferenceEquals(null, obj)) return false;**

**if (ReferenceEquals(this, obj)) return true;**

**if (obj.GetType() != this.GetType()) return false;**

**return Equals((Point)obj);}**

**//we override GetHashCode() on the Point , Line**

**public override int GetHashCode(){**

**unchecked{return (X \* 397) ^ Y;}}**

**public override string ToString(){**

**return $"({X}, {Y})";}}}**

**namespace AdapterCachingPro{**

**public class Line{**

**public Point Start;**

**public Point End;**

**public Line(Point start, Point end){**

**this.Start = start;**

**this.End = end;}**

**protected bool Equals(Line other){**

**return Equals(Start, other.Start) && Equals(End, other.End);}**

**//it will resolve the issue of the replicate temp data**

**//make check equal by reference if it equals return true**

**//and then check for the two Point Start , End**

**public override bool Equals(object obj){**

**if (ReferenceEquals(null, obj)) return false;**

**if (ReferenceEquals(this, obj)) return true;**

**if (obj.GetType() != this.GetType()) return false;**

**return Equals((Line)obj);}**

**//we override GetHashCode() on the Point , Line**

**public override int GetHashCode(){**

**unchecked{**

**return ((Start != null ? Start.GetHashCode() : 0) \* 397) ^ (End != null ? End.GetHashCode() : 0);}}}}**

**public abstract class VectorObject : Collection<Line>{}**

**public class VectorRectangle : VectorObject{**

**public VectorRectangle(int x, int y, int width, int height){**

**Add(new Line(new Point(x, y), new Point(x + width, y)));**

**Add(new Line(new Point(x + width, y), new Point(x + width, y + height)));**

**Add(new Line(new Point(x, y), new Point(x, y + height)));**

**Add(new Line(new Point(x, y + height), new Point(x + width, y + height)));}}**

**//on the Adapter class we inherit from IEnumerable to provide ability to override GetEnumerator to //get SelectMany(points)**

**using System;**

**using System.Collections;**

**using System.Collections.Generic;**

**using static System.Console;**

**namespace AdapterCachingPro{**

**public class LineToPointAdapter : IEnumerable<Point>{**

**private static int count = 0;**

**//by using cache which is used to store all the points with the hash code it refer**

**static Dictionary<int, List<Point>> cache = new Dictionary<int, List<Point>>();**

**private int hash;**

**public LineToPointAdapter(Line line){**

**//we check to the line GetHashCode() which return X,Y GetHashCode()**

**hash = line.GetHashCode();**

**// we already have it**

**if (cache.ContainsKey(hash))**

**return;**

**WriteLine($"{++count}: Generating points for line [{line.Start.X},{line.Start.Y}]-[{line.End.X},{line.End.Y}] (with caching)");**

**List<Point> points = new List<Point>();**

**int left = Math.Min(line.Start.X, line.End.X);**

**int right = Math.Max(line.Start.X, line.End.X);**

**int top = Math.Min(line.Start.Y, line.End.Y);**

**int bottom = Math.Max(line.Start.Y, line.End.Y);**

**int dx = right - left;**

**int dy = line.End.Y - line.Start.Y;**

**if (dx == 0){**

**for (int y = top; y <= bottom; ++y){points.Add(new Point(left, y));}}**

**else if (dy == 0){for (int x = left; x <= right; ++x){points.Add(new Point(x, top));}}**

**//on each line convert to point it will store to caching**

**cache.Add(hash, points);}**

**public IEnumerator<Point> GetEnumerator(){return cache[hash].GetEnumerator();}**

**IEnumerator IEnumerable.GetEnumerator(){return cache[hash].GetEnumerator();}}}**

**Lesson04 Adapter with DI**

**Notes:-**

**1-if you have case that if you have list of buttons with different command type and you want to pass different Meta data through it you use register adapter as below**

**namespace AdapterDIPro{**

**public interface ICommand{void Execute();}}**

**using System;**

**namespace AdapterDIPro{**

**public class OpenCommand : ICommand{**

**public void Execute(){Console.WriteLine("Opening a file");}}}**

**using System;**

**namespace AdapterDIPro{**

**public class SaveCommand : ICommand{**

**public void Execute(){Console.WriteLine("Saving current file");}}}**

**//in the class Button we inject ICommand as parameter**

**using System;**

**namespace AdapterDIPro{**

**public class Button{**

**private ICommand command;**

**private string name;**

**public Button(ICommand command, string name){**

**if (command == null){throw new ArgumentNullException(paramName: nameof(command));}**

**this.command = command;**

**this.name = name;}**

**public void Click(){command.Execute();}**

**public void PrintMe(){Console.WriteLine($"I am a button called {name}");}}}**

**//in the Editor class we inject IEnumerable<Button> as parameter**

**using System.Collections.Generic;**

**namespace AdapterDIPro{**

**public class Editor{**

**private readonly IEnumerable<Button> buttons;**

**public IEnumerable<Button> Buttons => buttons;**

**public Editor(IEnumerable<Button> buttons){this.buttons = buttons;}**

**public void ClickAll(){**

**foreach (var btn in buttons){btn.Click();}}}}**

**//on the main.cs in order to generate IEnumerable<Button> with generate two instance of Button //we have to use RegisterAdapter that will ensure to generate instance of every class inherit from //ICommand and we can pass meta data as below**

**class Program{**

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

**// for each ICommand, a ToolbarButton is created to wrap it, and all**

**// are passed to the editor**

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

**b.RegisterType<OpenCommand>()**

**.As<ICommand>()**

**.WithMetadata("Name", "Open");**

**b.RegisterType<SaveCommand>()**

**.As<ICommand>()**

**.WithMetadata("Name", "Save");**

**//b.RegisterType<Button>();**

**b.RegisterAdapter<ICommand, Button>(cmd => new Button(cmd, ""));**

**b.RegisterAdapter<Meta<ICommand>, Button>(cmd =>**

**new Button(cmd.Value, (string)cmd.Metadata["Name"]));**

**b.RegisterType<Editor>();**

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

**var editor = c.Resolve<Editor>();**

**editor.ClickAll();**

**// problem: only one button**

**foreach (var btn in editor.Buttons)**

**btn.PrintMe();}}}**