**Section09 Composite Design Pattern**

**Notes:-**

**1-Composite: Treating individual and aggregate objects uniformly**

**(Mechanism for treating individual (scalar) objects and compositions of objects in uniform manner)**

**(Composite design pattern is used to treat both single (scalar) and composite objects uniformly)**

**Motivation**

**1-Objects use other objects fields / properties / members through inheritance and composition**

**2-Composition lets us make compound objects**

**A-mathematical expression composed of simple expressions**

**B-grouping of shapes that consists of several shapes**

**Lesson01 Simple Composite Pattern**

**Notes:-**

**1-with using Composite Design pattern by make base class that all childs inherit from it and declare property inherit from that base class we can store single and group of child class as below**

**//by making base class contains property List of the same class type as below**

**namespace CompositePro{**

**public class Circle : GraphicObject{**

**public override string Name => "Circle";}}**

**namespace CompositePro{**

**public class Square : GraphicObject{**

**public override string Name => "Square";}}**

**using System;**

**using System.Collections.Generic;**

**using System.Text;**

**namespace CompositePro{**

**//in the base class we declare list of the GraphicObject**

**//which allow to store list of the same type inside the GraphicObject**

**public class GraphicObject{**

**public virtual string Name { get; set; } = "Group";**

**public string Color;**

**//you have to initialize the List in lazy loading way as below**

**private Lazy<List<GraphicObject>> children = new Lazy<List<GraphicObject>>();**

**public List<GraphicObject> Children => children.Value;**

**private void Print(StringBuilder sb, int depth){**

**sb.Append(new string('\*', depth))**

**.Append(string.IsNullOrWhiteSpace(Color) ? string.Empty : $"{Color} ")**

**.AppendLine($"{Name}");**

**foreach (var child in Children)**

**child.Print(sb, depth + 1);}**

**public override string ToString(){**

**var sb = new StringBuilder();**

**Print(sb, 0);**

**return sb.ToString();}}}**

**using static System.Console;**

**namespace CompositePro{**

**class Program{**

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

**var drawing = new GraphicObject { Name = "My Drawing" };**

**drawing.Children.Add(new Square { Color = "Red" });**

**drawing.Children.Add(new Circle { Color = "Yellow" });**

**//we create another GraphicObject and added as new item on the drawing object**

**var group = new GraphicObject();**

**group.Children.Add(new Circle { Color = "Blue" });**

**group.Children.Add(new Square { Color = "Blue" });**

**drawing.Children.Add(group);**

**WriteLine(drawing);}}}**

**Lesson02 Neural Networks**

**Notes:-**

**1-if we want to combine group of class with another group of another class inherit of the same class we can also using the Composite design pattern as below**

**using System.Collections.ObjectModel;**

**namespace NeuronPro{**

**//we make class inherit from the Collection<Neuron>**

**public class NeuronLayer : Collection<Neuron>{}}**

**using System;**

**using System.Collections;**

**using System.Collections.Generic;**

**namespace NeuronPro{**

**//Neuron contains List<Neuron> of In , Out**

**public class Neuron : IEnumerable<Neuron>{**

**public float Value;**

**private Lazy<List<Neuron>> \_out = new Lazy<List<Neuron>>();**

**private Lazy<List<Neuron>> \_in2 = new Lazy<List<Neuron>>();**

**public List<Neuron> In => \_in2.Value;**

**public List<Neuron> Out => \_out.Value;**

**public IEnumerator<Neuron> GetEnumerator(){yield return this;}**

**IEnumerator IEnumerable.GetEnumerator(){yield return this;}}}**

**using System.Collections.Generic;**

**namespace NeuronPro{**

**//this extension is make composite between IEnumerable<Neuron> of source and target**

**public static class ExtensionMethods{**

**public static void ConnectTo(this IEnumerable<Neuron> self, IEnumerable<Neuron> other){**

**if (ReferenceEquals(self, other)) return;**

**foreach (var from in self)**

**foreach (var to in other){**

**from.Out.Add(to);**

**to.In.Add(from);}}}}**

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

**var neuron1 = new Neuron();**

**var neuron2 = new Neuron();**

**var layer1 = new NeuronLayer();**

**var layer2 = new NeuronLayer();**

**//we make combination between two instances and with also different class that inherit the same //Neuron class**

**neuron1.ConnectTo(neuron2);**

**neuron1.ConnectTo(layer1);**

**layer1.ConnectTo(layer2);**

**ReadLine();}**

**Lesson03 Composite Specification**

**Notes:-**

**1-we can apply composite filter specification in generic way as below**

**namespace CompositeSpecPro.Enums{**

**public enum Color{Red,Green,Blue}**

**public enum Size{Small,Meduim,Large,Yuge}}**

**//the abstract class is the base class for all filters**

**namespace CompositeSpecPro.Interfaces{**

**public abstract class ISpecification<T>{public abstract bool IsSatisfied(T obj);}}**

**//Size Specification inherit from ISpecification**

**using CompositeSpecPro.Enums;**

**using CompositeSpecPro.Models;**

**namespace CompositeSpecPro.Interfaces.Filters{**

**public class SizeSpecification : ISpecification<Product>{**

**public Size \_size { get; set; }**

**public SizeSpecification(Size size) { this.\_size = size; }**

**public override bool IsSatisfied(Product obj){return obj.Size.Equals(\_size);}}}**

**//Color Specification inherit from ISpecification**

**using CompositeSpecPro.Enums;**

**using CompositeSpecPro.Models;**

**namespace CompositeSpecPro.Interfaces.Filters{**

**public class ColorSpecification : ISpecification<Product>{**

**public Color \_color { get; set; }**

**public ColorSpecification(Color color) { this.\_color = color; }**

**public override bool IsSatisfied(Product obj){return obj.Color == \_color;}}}**

**//Composite Specification inherit from ISpecification take params ISpecification**

**namespace CompositeSpecPro.Interfaces.Filters{**

**//CompositeSpecification make params Ispcification<T>[] items**

**public abstract class CompositeSpecification<T> : ISpecification<T>{**

**protected readonly ISpecification<T>[] Items;**

**public CompositeSpecification(params ISpecification<T>[] items){Items = items;}}}**

**using System;**

**using System.Linq;**

**namespace CompositeSpecPro.Interfaces.Filters{**

**//combinator inherit from CompositeSpecification**

**//so we can pass as we want of specification filters as below**

**public class AndSpecification<T> : CompositeSpecification<T>{**

**public AndSpecification(params ISpecification<T>[] items) :base(items){}**

**public override bool IsSatisfied(T t){**

**//Any => Or Specification**

**return Items.All(i => i.IsSatisfied(t));}}}**

**using System.Collections.Generic;**

**namespace CompositeSpecPro.Interfaces{**

**//this interface takes items and Ispecification as parmaeters**

**public interface IFilter<T>{**

**IEnumerable<T> Filter(IEnumerable<T> items, ISpecification<T> sepc);}}**

**using CompositeSpecPro.Models;**

**using System.Collections.Generic;**

**//it will take Ispecification as parameters so we can pass Composite filter, Color filter, Size filter**

**namespace CompositeSpecPro.Interfaces.Filters{**

**public class BetterFilter : IFilter<Product>{**

**public IEnumerable<Product> Filter(IEnumerable<Product> items, ISpecification<Product> spec){**

**foreach (var item in items){**

**if (spec.IsSatisfied(item))**

**yield return item;}}}**

**//on the main entry we make apply the composite filters by pass as we want of I Specification filters //as below**

**using CompositeSpecPro.Enums;**

**using CompositeSpecPro.Filters;**

**using CompositeSpecPro.Interfaces.Filters;**

**using CompositeSpecPro.Models;**

**using System;**

**using System.Collections.Generic;**

**namespace CompositeSpecPro{**

**class Program{**

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

**var apple = new Product("Apple", Color.Green, Size.Small);**

**var tree = new Product("Tree", Color.Green, Size.Meduim);**

**var house = new Product("House", Color.Blue, Size.Large);**

**var prods = new List<Product>() { apple, tree, house };**

**//each time that we need new filter feature, we need to modify new filter feature in the , so //the solution is used with Open Closed Design Pattern**

**var pf = new ProductFilter();**

**Console.WriteLine("Green Products (old)");**

**foreach (var p in pf.FilterByColor(prods, Color.Green)) { Console.WriteLine(p.Name); }**

**var bf = new BetterFilter();**

**Console.WriteLine("Better Filter");**

**foreach (var p in bf.Filter(prods, new ColorSpecification(Color.Green))){**

**Console.WriteLine(p.Name);}**

**Console.WriteLine("With Multiple Better Filter");**

**foreach (var p in bf.Filter(prods, new AndSpecification<Product>(new ColorSpecification(Color.Green), new SizeSpecification(Size.Meduim)))){**

**Console.WriteLine(p.Name);}**

**Console.ReadLine();}}}**