Компоновщик

Подготовил доклад:
Воронин Никита 341 группа.

Что такое компоновщик ?

- **Компоновщик** (англ. *Composite pattern*) — структурный шаблон проектирования, объединяющий объекты в древовидную структуру для представления иерархии от частного к целому. Компоновщик позволяет клиентам обращаться к отдельным объектам и к группам объектов одинаково.

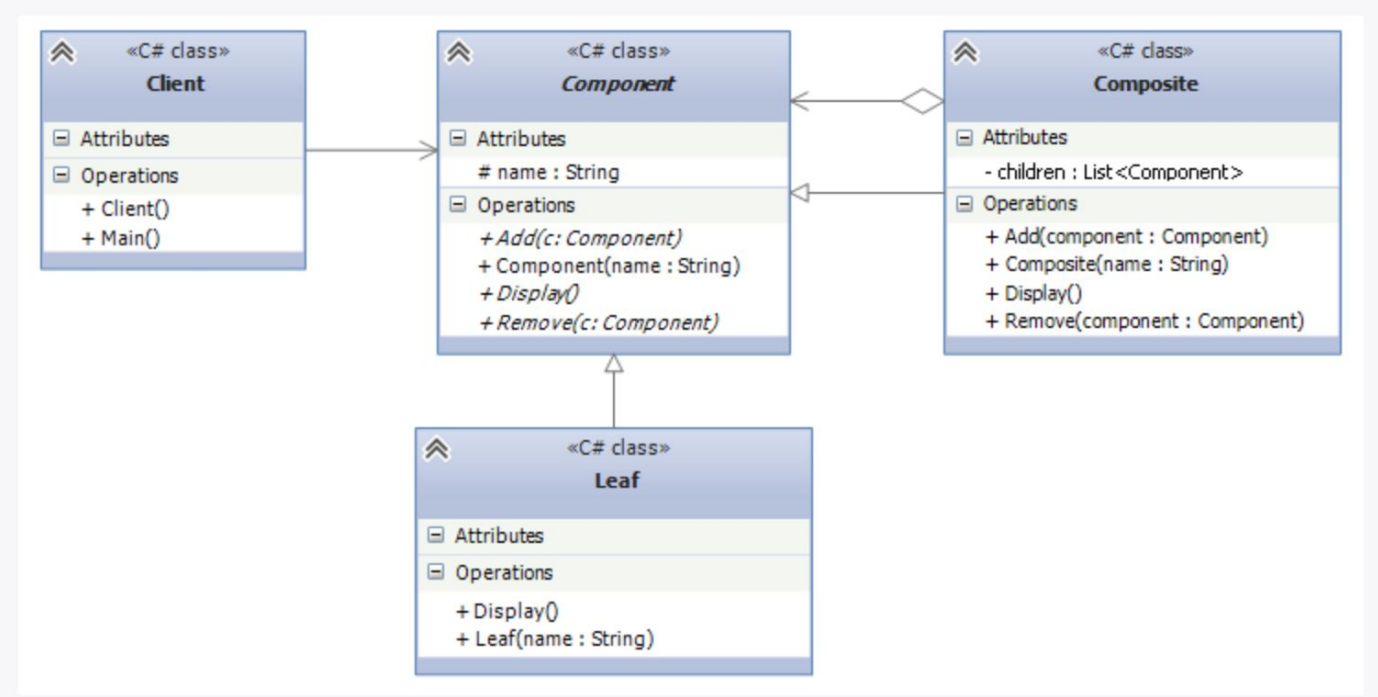
- Образно реализацию паттерна можно представить в виде меню, которое имеет различные пункты. Эти пункты могут содержать подменю, в которых, в свою очередь, также имеются пункты. То есть пункт меню служит с одной стороны частью меню, а с другой стороны еще одним меню. В итоге мы однообразно можем работать как с пунктом меню, так и со всем меню в целом.

Плюсы и минусы

- **Достоинства паттерна Composite**
- В систему легко добавлять новые примитивные или составные объекты, так как паттерн Composite использует общий базовый класс Component.
- Код клиента имеет простую структуру – примитивные и составные объекты обрабатываются одинаковым образом.
- Паттерн Composite позволяет легко обойти все узлы древовидной структуры
- **Недостатки паттерна Composite**
- Неудобно осуществить запрет на добавление в составной объект Composite объектов определенных типов. Так, например, в состав римской армии не могут входить боевые слоны.

Когда использовать компоновщик?

- Когда объекты должны быть реализованы в виде иерархической древовидной структуры
- Когда клиенты единообразно должны управлять как целыми объектами, так и их составными частями. То есть целое и его части должны реализовать один и тот же интерфейс
- С помощью UML шаблон компоновщик можно представить следующим образом:



Формальное определение паттерна на C# могло бы выглядеть так:

```
class Client
{
    public void Main()
    {
        Component root = new Composite("Root");
        Component leaf = new Leaf("Leaf");
        Composite subtree = new Composite("Subtree");
        root.Add(leaf);
        root.Add(subtree);
        root.Display();
    }
}
abstract class Component
{
    protected string name;

    public Component(string name)
    {
        this.name = name;
    }
    public abstract void Display();
    public abstract void Add(Component c);
    public abstract void Remove(Component c);
}
class Composite : Component
{
    List<Component> children = new List<Component>();

    public Composite(string name)
        : base(name)
    {}

    public override void Add(Component component)
    {
        children.Add(component);
    }

    public override void Remove(Component component)
    {
        children.Remove(component);
    }
```

```
    public override void Display()
    {
        Console.WriteLine(name);

        foreach (Component component in children)
        {
            component.Display();
        }
    }
}
class Leaf : Component
{
    public Leaf(string name)
        : base(name)
    {}

    public override void Display()
    {
        Console.WriteLine(name);
    }
    public override void Display()
    {
        Console.WriteLine(name);
    }

    public override void Add(Component component)
    {
        throw new NotImplementedException();
    }

    public override void Remove(Component component)
    {
        throw new NotImplementedException();
    }
}
```

Участники

- **Component**: определяет интерфейс для всех компонентов в древовидной структуре
- **Composite**: представляет компонент, который может содержать другие компоненты и реализует механизм для их добавления и удаления
- **Leaf**: представляет отдельный компонент, который не может содержать другие компоненты
- **Client**: клиент, который использует компоненты

- Пример. Нужно создать объект файловой системы. Файловую систему составляют папки и файлы. Каждая папка также может включать в себя папки и файлы. То есть получается древовидная иерархическая структура, где с вложенными папками нам надо работать также, как и с папками, которые их содержат. Для реализации данной задачи и воспользуемся паттерном Компоновщик:

```
class Program
{
    static void Main(string[] args)
    {
        Component fileSystem = new Directory("Файловая система");
        // определяем новый диск
        Component diskC = new Directory("Диск C");
        // новые файлы
        Component pngFile = new File("12345.png");
        Component docxFile = new File("Document.docx");
        // добавляем файлы на диск C
        diskC.Add(pngFile);
        diskC.Add(docxFile);
        // добавляем диск C в файловую систему
        fileSystem.Add(diskC);
        // выводим все данные
        fileSystem.Print();
        Console.WriteLine();
        // удаляем с диска C файл
        diskC.Remove(pngFile);
        // создаем новую папку
        Component docsFolder = new Directory("Мои Документы");
        // добавляем в нее файлы
        Component txtFile = new File("readme.txt");
        Component csFile = new File("Program.cs");
        docsFolder.Add(txtFile);
        docsFolder.Add(csFile);
        diskC.Add(docsFolder);

        fileSystem.Print();

        Console.Read();
    }
}

abstract class Component
{
    protected string name;

    public Component(string name)
    {
        this.name = name;
```

```
    public virtual void Add(Component component){}

    public virtual void Remove(Component component) { }

    public virtual void Print()
    {
        Console.WriteLine(name);
    }
}
class Directory :Component
{
    private List<Component> components = new List<Component>();

    public Directory(string name)
        : base(name)
    {
    }

    public override void Add(Component component)
    {
        components.Add(component);
    }

    public override void Remove(Component component)
    {
        components.Remove(component);
    }

    public override void Print()
    {
        Console.WriteLine("Узел " + name);
        Console.WriteLine("Подузлы:");
        for(int i=0; i<components.Count;i++)
        {
            components[i].Print();
        }
    }
}

class File : Component
{
    public File(string name)
            : base(name)
    {}
```

- В итоге подобная система обладает неплохой гибкостью: если мы захотим добавить новый вид компонентов, нам достаточно унаследовать новый класс от Component.
- И также применяя компоновщик, мы легко можем обойти все узлы древовидной структуры.

Спасибо за внимание!