Лабораторная работа №8
Изучение и практическое применение структурного шаблона проектирования Декоратор

Декоратор — это структурный паттерн проектирования, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».

Шаблон Декоратор предоставляет гибкую альтернативу практике создания подклассов с целью расширения функциональности.

Известен также под менее распространённым названием Обёртка (Wrapper), которое во многом раскрывает суть реализации шаблона.

Основная задача шаблона

Объект, который предполагается использовать, выполняет основные функции. Однако может потребоваться добавить к нему некоторую дополнительную функциональность, которая будет выполняться *до*, *после* или даже *вместо* основной функциональности объекта.

Одним из основных способов расширения функционала классов является наследование. Но механизм наследования имеет несколько досадных проблем:

Нет возможности изменить поведение существующего объекта. Для этого вам надо создать новый объект, выбрав другой подкласс.

Невозможно выполнить наследовать поведения нескольких классов одновременно. Из-за этого приходится создавать множество подклассов-комбинаций для получения совмещённого поведения.

Декоратор предусматривает расширение функциональности объекта без определения подклассов. Это возможно из-за того, что в шаблоне используется *агрегация* (*композиция*). Это когда один объект содержит ссылку на другой и делегирует ему работу, вместо того чтобы самому наследовать его поведение. Именно на этом принципе построен паттерн Декоратор.

Абстрактный (структурный) пример

Базовый интерфейс Component определяет поведение, которое изменяется декораторами.

```
public abstract class Component
{
    public abstract string Operation();
}
```

Классы ConcreteComponent предоставляют реализации поведения по умолчанию. Может быть несколько вариаций этих классов.

```
class ConcreteComponent : Component
{
    public override string Operation()
    {
        return "ConcreteComponent";
    }
}
```

Базовый класс Decorator следует тому же интерфейсу, что и другие компоненты. Основная цель этого класса - определить интерфейс обёртки для всех конкретных декораторов. Реализация кода обёртки по умолчанию может включать в себя поле для хранения завёрнутого компонента и средства его инициализации.

```
abstract class Decorator : Component
{
    protected Component _component;

    public Decorator(Component component)
    {
        this._component = component;
    }

    public void SetComponent(Component component)
    {
        this._component = component;
    }

    // Декоратор делегирует всю работу обёрнутому компоненту.
    public override string Operation()
    {
        if (this._component != null)
        {
            return this._component.Operation();
        }
        else
        {
            return string.Empty;
        }
    }
}
```

Конкретные Декораторы вызывают обёрнутый объект и изменяют его результат некоторым образом.

```
class ConcreteDecoratorA : Decorator
```

```
    {
        public ConcreteDecoratorA(Component comp) : base(comp)
        {
        }

        // Декораторы могут вызывать родительскую реализацию операции, вместо
        // того, чтобы вызвать обёрнутый объект напрямую. Такой подход
упрощает
        // расширение классов декораторов.
        public override string Operation()
        {
            return $"ConcreteDecoratorA({base.Operation()})";
        }
    }
```

Декораторы могут выполнять своё поведение до или после вызова обёрнутого объекта.

```
    class ConcreteDecoratorB : Decorator
    {
        public ConcreteDecoratorB(Component comp) : base(comp)
        {
        }

        public override string Operation()
        {
            return $"ConcreteDecoratorB({base.Operation()})";
        }
    }
```

Клиентский код работает со всеми объектами, используя интерфейс Component. Таким образом, он остаётся независимым от конкретных классов компонентов, с которыми работает.

```
    public class Client
    {
        public void ClientCode(Component component)
        {
            Console.WriteLine("RESULT: " + component.Operation());
        }
    }
```

Пример использования данных декораторов приведен ниже. Обратите внимание, что декораторы могут обёртывать не только компоненты, но и другие декораторы. Таким образом, достигается расширение реализованных операций декоратором.

```
static void Main(string[] args)
{
    Client client = new Client();
```

```
    var simple = new ConcreteComponent();
    Console.WriteLine("Client: I get a simple component:");
    client.ClientCode(simple);
    Console.WriteLine();

    ConcreteDecoratorA decorator1 = new ConcreteDecoratorA(simple);
    ConcreteDecoratorB decorator2 = new ConcreteDecoratorB(decorator1);
    Console.WriteLine("Client: Now I've got a decorated component:");
    client.ClientCode(decorator2);
}
```

Пример расширения классов StreamReader и StreamWriter

Допустим имеется задача чтения и записи текстовых файлов. К примеру, разрабатываем систему ведения логов. При записи данных в текстовый файл традиционно используются классы SreamWriter, для чтения текстовых файлов – StreamReader.

Требуется при считывании строк добавлять ее номер в начало. Т.е. при считывании строки «Это первая строка» из файла, из функции ReadLine возвращалось «1. Это первая строка». И так для всех считываемых строк.

То же самое нужно сделать при записи строк в текстовый файл.

Первым, что приходит в голову – это просто добавлять в прикладном (клиентском) коде к каждой записываемой строке номер. Однако, здесь может быть обнаружены следующие ошибки и неудобства:

1. Это просто можно будет забыть сделать. Т.е. программист всегда должен об этом помнить.
2. Нумерация должно быть в пределах каждого файла. Теперь представьте себе, если мы одновременно открыли и пишем пять файлов: лог ошибок, лог предупреждений, лог действий и др. Нам нужно пять разных счетчиков номеров строк. Это опять неудобно и нарушает базовый принцип ООП – инкапсуляцию.

Поэтому наиболее верным способом решения данной проблемы является расширение классов StreamReader и StreamWriter. Использование механизма наследования решит поставленную проблему. Однако это решение лишит возможности масштабирования задачи. Например, если в последующем нужно будет в некоторых файлах автоматически записывать строку на двух языках (на русском и английском, например). Нужно будет сделать еще один класс. И так далее, могут просто получиться десятки классов.

Поэтому, наиболее подходящим способом расширения функционирования классов будет создание обертки над классами

StreamReader и StreamWriter, а именно, просто переопределения функции ReadLine и WriteLine – создание декораторов.

Разработка декораторов для класса StreamReader

Для начала создадим шаблонный класс декоратор:

```
public abstract class TextReaderDecorator : TextReader
{
    protected TextReader component;

    public TextReaderDecorator(TextReader component)
    {
        this.component = component;
    }
    public override string ReadLine()
    {
        if(component != null)
        {
            return component.ReadLine();
        }

        return "Decorator.ReadLine()";
    }
}
```

Обратите внимание на следующие особенности:

1. Наследование выполняется не от класса StreamReader, а от абстрактного класса TextReader. Именно от него же и наследован класс StreamReader. Т.е. мы декорируем базовый класс для большей гибкости применения декоратора.
2. Основным и единственным полем класса является ссылка на тот же самый базовый класс. Именно по этой ссылке и будут делегироватся основные вызовы методов.

Теперь конкретный декоратор, который добавляет номер строки при чтении каждой строки из файла:

```
public class StreamReaderDecoratorNumLines : TextReaderDecorator
{
    private int CurrentLineNumber;
    public StreamReaderDecoratorNumLines(TextReader component) :
base(component)
    {

    }

    public override string ReadLine()
    {
```

```
            string lineFromBase = base.ReadLine();
            return lineFromBase == null ? null : $"{CurrentLineNumber++}. " +
lineFromBase;
        }
    }
```

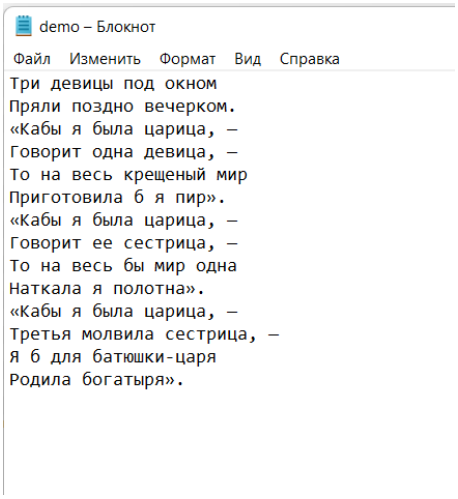
Для демонстрации иерархического декорирования создадим еще один декоратор, который просто приводит считанную строку к нижнему регистру:

```
    public class StreamReaderDecoratorLowerCase : TextReaderDecorator
    {
        public StreamReaderDecoratorLowerCase(TextReader component) :
base(component)
        {

        }
        public override string ReadLine()
        {
            string lineFromBase = base.ReadLine();
            return lineFromBase == null ? null : lineFromBase.ToLower();
        }
    }
```

Теперь в классе Program можно продемонстрировать эти декораторы.

Тестирование чтения файла будет выполнять не следующем текстовом файле:



При создании экземпляра класса StreamReader без декорирования использовался бы следующий код:

```
string path = @"C:\Intel\demo.txt";

Console.WriteLine("******считываем построчно********");
TextReader reader = new StreamReader(path, System.Text.Encoding.Default);

string line;
```

```
while ((line = reader.ReadLine()) != null)
{
    Console.WriteLine(line);
}
```

При этом файл бы вывелся на консоль в том виде, в котором он записан в файле. А при использовании декораторов, мы созданный экземпляр упаковываем в класс декоратор:

```
string path = @"C:\Intel\demo.txt";

Console.WriteLine("******считываем построчно********");
TextReader reader =  new StreamReaderDecoratorNumLines(
                    new StreamReader(path, System.Text.Encoding.Default)));

string line;
while ((line = reader.ReadLine()) != null)
{
    Console.WriteLine(line);
}
```

Здесь использовали декоратор, добавляющий номера строк.

Однако, прелесть декораторов в том, что мы одновременно можем декорировать несколькими декораторами:

```
TextReader reader = new StreamReaderDecoratorLowerCase(
    new StreamReaderDecoratorNumLines(
         new StreamReader(path, System.Text.Encoding.Default)));
```

Задание на лабораторную работу:

1. Реализовать примеры и методических указаний.
2. Реализовать декоратор над классом TextReader, добавляющий к считанной строке текущее время и дату в начало строки. Продемонстрировать работу всех трех декораторов на одном экземпляре.
3. Реализовать декораторы над классом TextWriter (принцип точно такой же как и с классом TextReader). Один декоратор добавляет номер записываемой строки, другой добавляет текущее время и дату начало записываемой строки. Продемонстрировать работы декораторов как по отдельности, так и вместе.