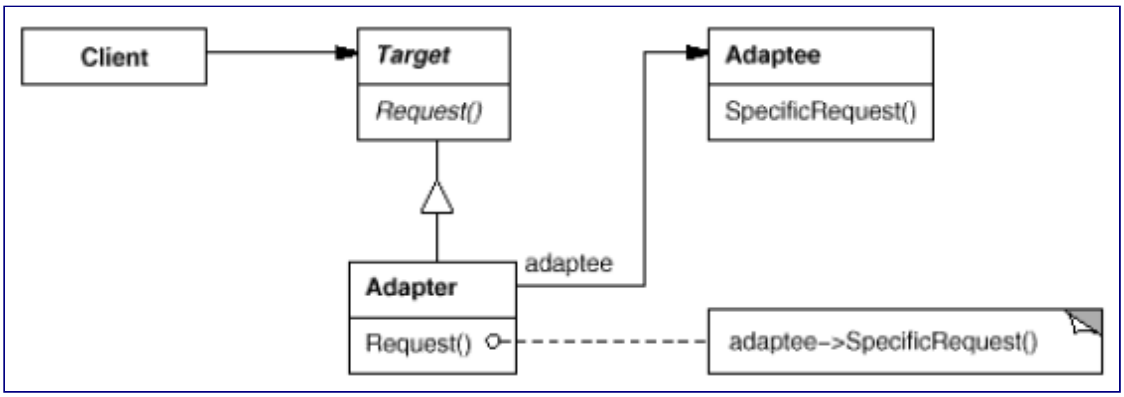
Структурные паттерны

М. Э. Абрамян, 2021

Adapter, Composite, Decorator

OOP2Struc1°.



Adapter (**Адаптер**) — структурный паттерн.

Известен также под именем **Wrapper** (**Обертка**).

Частота использования: выше средней.

Назначение: преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты. Адаптер обеспечивает совместную работу классов с несовместимыми интерфейсами, которая без него была бы невозможна.

Участники:

• *Target* (*Целевой класс*) — определяет зависящий от предметной области интерфейс, которым пользуется Client;

• *Client* (*Клиент*) — вступает во взаимоотношения с объектами, удовлетворяющими интерфейсу Target;

• *Adaptee* (*Адаптируемый класс*) — определяет существующий интерфейс, который нуждается в адаптации;

• *Adapter* (*Адаптер*) — адаптирует интерфейс Adaptee к интерфейсу Target.

В данном задании рассматривается вариант адаптера, использующий композицию. Такой вариант называется *адаптером объекта*, поскольку адаптируется *объект* типа Adaptee, входящий в виде ссылочного поля в класс Adapter.

Задание 1. Абстрактный класс Target содержит три абстрактных метода: GetA, GetB и Request (не имеют параметров, возвращают значение целого типа). Класс ConcreteTarget является потомком класса Target; он содержит поля a и b целого типа, которые инициализируются в конструкторе, имеющем одноименные параметры. Методы GetA и GetB класса ConcreteTarget возвращают значения полей a и b соответственно, а метод Request возвращает сумму этих полей.

Класс Adaptee содержит два целочисленных поля a и b, конструктор с параметрами a и b, задающий значения этих полей, метод GetAll, возвращающий текущие значения полей (либо с помощью выходных параметров, либо с помощью структурного возвращаемого значения — массива или кортежа), и метод SpecificRequest без параметров, возвращающий произведение полей a и b.

Реализовать класс Adapter, адаптирующий класс Adaptee к интерфейсу класса Target. Класс должен быть адаптером объекта: он порождается от класса Target и включает ссылку ad на экземпляр адаптируемого объекта Adaptee. Ссылка ad инициализируется в конструкторе класса Adapter с помощью одноименного параметра (других параметров конструктор не имеет). Метод Request класса Adapter должен вызывать метод SpecificRequest объекта ad, а методы GetA и GetB — возвращать значения полей a и b объекта ad, используя его метод GetAll.

Дано целое число N (≤ 6) и набор из N троек (C, A, B), где C является символом "+" или "*", а элементы A и B являются целыми числами. Создать структуру данных (например, массив)

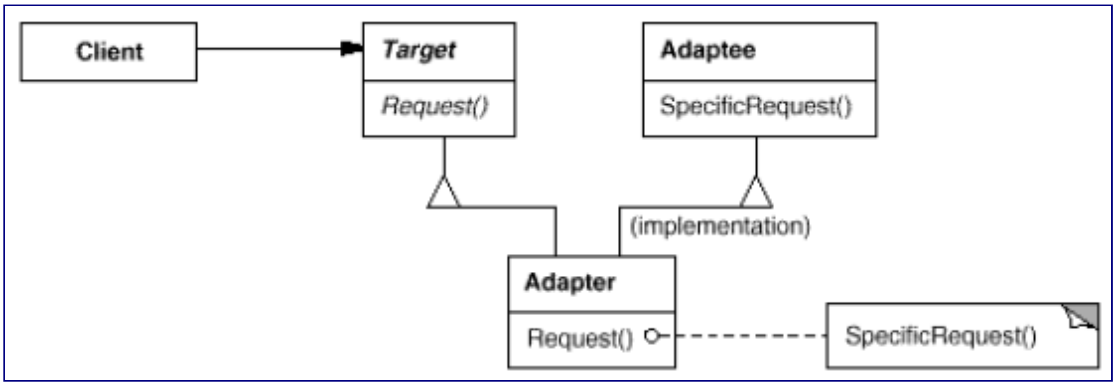
с элементами-ссылками типа Target и заполнить ее объектами типа ConcreteTarget (для троек с символом "+") и Adapter (для троек с символом "*") с полями, равными A и B. Перебирая элементы полученного набора в обратном порядке, вывести для каждого из них значения полей a, b и результат выполнения метода Request.

```
public class Adaptee
{
    // Do not change the implementation of the class
    int a, b;
    public Adaptee(int a, int b)
    {
        this.a = a;
        this.b = b;
    }
    public void GetAll(out int a, out int b)
    {
        a = this.a;
        b = this.b;
    }
    public int SpecificRequest()
    {
        return a * b;
    }
}

public abstract class Target
{
    public abstract int GetA();
    public abstract int GetB();
    public abstract int Request();
}

// Implement the ConcreteTarget and Adapter classes
```

OOP2Struc2°.



Adapter (**Адаптер**) — структурный паттерн.

В данном задании рассматривается вариант адаптера, использующий множественное наследование. Такой вариант называется *адаптером класса*, поскольку класс Adapter порождается от *класса* Adaptee, наследуя его реализацию (а также от класса Target, наследуя его интерфейс). Если язык не поддерживает множественное наследование, то адаптер класса можно реализовать с помощью *интерфейсов*; для этого надо преобразовать абстрактный класс Target в интерфейс ITarget, сделать класс Adapter потомком класса Adaptee и добавить к нему интерфейс ITarget.

Задание 2. Выполнить предыдущее задание (см. OOP2Struc1), реализовав Adapter как адаптер класса. Для языков, не поддерживающих множественное наследование, определить интерфейс ITarget с методами GetA, GetB и Request и добавить интерфейс ITarget к классам ConcreteTarget и Adapter (абстрактный класс Target при этом не требуется; для хранения исходных данных надо использовать структуру с элементами интерфейсного типа ITarget).

```
public class Adaptee
{
    // Do not change the implementation of the class
    int a, b;
    public Adaptee(int a, int b)
```

```
        {
            this.a = a;
            this.b = b;
        }
        public void GetAll(out int a, out int b)
        {
            a = this.a;
            b = this.b;
        }
        public int SpecificRequest()
        {
            return a * b;
        }
    }

    public abstract class Target
    {
        // Convert this abstract class into the ITarget interface
        public abstract int GetA();
        public abstract int GetB();
        public abstract int Request();
    }

    // Implement the ConcreteTarget and Adapter classes
```

OOP2Struc3°. **Adapter** (**Адаптер**) — структурный паттерн.

Задание 3. Дан абстрактный класс Shape, предоставляющий интерфейс для графических объектов: метод GetInfo без параметров, возвращающий строку с именем объекта и координатами левой верхней и правой нижней вершины ограничивающего прямоугольника (считается, что ось *OY* направлена вниз), и метод MoveBy(a, b) с двумя целочисленными параметрами, определяющими вектор, на который надо сместить данный графический объект (метод не возвращает значений). В классе Shape методы GetInfo и MoveBy являются абстрактными.

Также дан конкретный класс RectShape — потомок класса Shape, реализующий прямоугольник и имеющий конструктор с параметрами (x1, y1, x2, y2), задающими координаты левой верхней и правой нижней вершины этого прямоугольника. Метод GetInfo для данного класса возвращает строку вида "R(x1,y1)(x2,y2)" с текущими значениями координат.

Дан класс TextView для работы с текстовыми объектами. Он содержит поля x, y (координаты точки привязки — левого верхнего угла текстовой области), width, height (ширина и высота текстовой области) и методы GetOrigin (возвращает координаты точки привязки), SetOrigin (изменяет точку привязки), GetSize (возвращает размеры текстовой области) и SetSize (изменяет размеры текстовой области). Методы GetOrigin и GetSize возвращают результаты либо с помощью выходных параметров, либо с помощью структурного возвращаемого значения — массива или кортежа. Конструктор класса не имеет параметров (поля x и y полагаются равными 0, поля width и height — равными 1).

Реализовать класс TextShape, адаптирующий класс TextView к интерфейсу класса Shape. Класс должен быть адаптером объекта: он порождается от класса Shape и включает поле tview, являющееся ссылкой на экземпляр адаптируемого объекта TextView (других полей класс TextShape не содержит). Метод GetInfo класса TextShape должен возвращать строку вида "T(x1,y1)(x2,y2)" с текущими значениями левой верхней и правой нижней вершины ограничивающего прямоугольника. Включить в класс TextShape конструктор с параметрами (tview, x1, y1, x2, y2); смысл четырех последних параметров аналогичен смыслу параметров конструктора класса RectShape.

Дано целое число N (≤ 8) и набор из N пятерок (C, X1, Y1, X2, Y2), где C является символом "R" или "T", а остальные элементы являются целыми числами. Кроме того, даны целые числа A и B. Создать структуру данных (например, массив) с элементами типа Shape и заполнить ее объектами типа RectShape (для пятерок с символом "R") и TextShape (для пятерок с символом "T"), используя значения X1, Y1, X2, Y2 в качестве параметров соответствующего конструктора. Применить к каждому элементу созданного набора метод

MoveBy с параметрами A и B и вывести строковые представления элементов набора с помощью метода GetInfo (перебирая элементы в исходном порядке).

```
public class TextView
{
    // Do not change the implementation of the class
    int x, y;
    int width = 1, height = 1;
    public void GetOrigin(out int x, out int y)
    {
        x = this.x;
        y = this.y;
    }
    public void SetOrigin(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public void GetSize(out int width, out int height)
    {
        width = this.width;
        height = this.height;
    }
    public void SetSize(int width, int height)
    {
        this.width = width;
        this.height = height;
    }
}

public abstract class Shape
{
    public abstract string GetInfo();
    public abstract void MoveBy(int a, int b);
}

// Implement the RectShape and TextShape descendant classes
```

OOP2Struc4°. **Adapter** (**Адаптер**) — структурный паттерн.

Задание 4. Выполнить предыдущее задание (см. OOP2Struc3), реализовав TextShape как адаптер класса. Для языков, не поддерживающих множественное наследование, преобразовать абстрактный класс Shape в интерфейс IShape и добавить его к классам RectShape и TextShape (для хранения исходных данных в этом случае надо использовать структуру с элементами интерфейсного типа IShape).

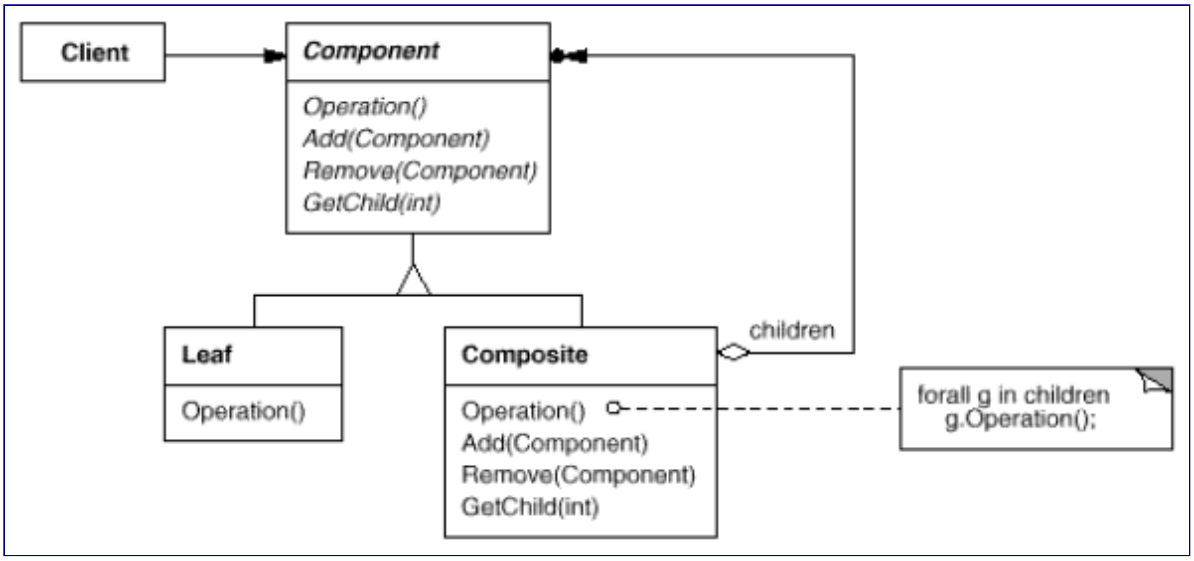
```
public class TextView
{
    // Do not change the implementation of the class
    int x, y;
    int width = 1, height = 1;
    public void GetOrigin(out int x, out int y)
    {
        x = this.x;
        y = this.y;
    }
    public void SetOrigin(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public void GetSize(out int width, out int height)
    {
        width = this.width;
        height = this.height;
    }
    public void SetSize(int width, int height)
    {
        this.width = width;
        this.height = height;
    }
```

```
}

public abstract class Shape
{
    // Convert this abstract class into the IShape interface
    public abstract string GetInfo();
    public abstract void MoveBy(int a, int b);
}

// Implement the RectShape and TextShape classes
// These classes must implement the IShape interface;
//   the TextShape class must be a descendant of the TextView class
```

OOP2Struc5°.



Composite (**Компоновщик**) — структурный паттерн.

Частота использования: выше средней.

Назначение: компонует объекты в древовидные структуры для представления иерархий "часть-целое". Позволяет клиентам единообразно трактовать индивидуальные и составные объекты.

Участники:

• *Component* (*Компонент*) — объявляет интерфейс для компонуемых объектов; предоставляет подходящую реализацию операций по умолчанию, общую для всех классов; объявляет интерфейс для доступа к дочерним компонентам и управления ими;

• *Leaf* (*Лист*) — представляет листовой узел композиции, не имеющий потомков;

• *Composite* (*Составной объект*) — хранит дочерние компоненты составного узла; реализует относящиеся к управлению потомками операции в интерфейсе класса Component;

• *Client* (*Клиент*) — манипулирует объектами композиции через интерфейс Component.

Объекты-листы и составные объекты могут также содержать дополнительные методы, определяющие их особое поведение, однако паттерн Composite ориентирован прежде всего на применение тех методов, которые являются общими у всех объектов композиции (хотя и реализуются по-разному для листов и составных объектов).

Задание 1. Реализовать иерархию классов, включающую абстрактный класс Component с методами Add и Show и конкретные классы Leaf и Composite. Метод Add с параметром-ссылкой с типа Component добавляет компонент с в набор дочерних компонентов (имеет смысл только для класса Composite; для класса Leaf не выполняет никаких действий), метод Show возвращает строковое представление данного компонента и всех его потомков (при наличии). В классе Component метод Add не выполняет никаких действий, а метод Show является абстрактным.

Классы Composite и Leaf содержат символьное поле data; метод Show класса Leaf возвращает это поле (преобразованное к строковому типу), метод Show класса Composite возвращает строку, начинающуюся с символа data, после которого стоит открывающая круглая скобка "(", затем указываются данные, полученные методом Show для каждого

дочернего компонента (без пробелов), а затем указывается закрывающая круглая скобка ")". Класс Composite хранит свои дочерние компоненты в виде массива или другой структуры с элементами-ссылками типа Component (можно считать, что любой объект типа Composite содержит не более 15 дочерних компонентов). Конструктор классов Leaf и Composite содержит один параметр — символ data.

Дана строка, содержащая не более 15 прописных и строчных латинских букв и набор согласованных круглых скобок, причем скобка "(" располагается после каждой прописной буквы (а в других позициях строки располагаться не может). Эта строка определяет набор данных для объектов типа Component; круглые скобки задают композицию этих объектов. Например, строка "aB(C(de)f)G()h" задает листья a, d, e, f, h и составные объекты B, C, G; при этом объект B содержит дочерние объекты C и f, объект C содержит дочерние объекты d и e, а объект G не содержит дочерних объектов (хотя и является составным). Создать все компоненты, определяемые данной строкой, и вывести для каждого из них описание, используя метод Show и перебирая компоненты в порядке их появления в исходной строке.

Указание. Для создания требуемых объектов можно использовать рекурсивную функцию Process(s, ind, parent), где s — исходная строка, ind — индекс анализируемого символа в строке, parent (типа Composite) — текущий родительский компонент (может быть пустым). Функция строит деревья компонентов и одновременно добавляет каждый создаваемый компонент в результирующую структуру, которая будет использована для вывода его описания.

```
public abstract class Component
{
    public virtual void Add(Component c) {}
    public abstract string Show();
}

// Implement the Left and Composite descendant classes
```

OOP2Struc6°. **Composite** (**Компоновщик**) — структурный паттерн.

Задание 2. Реализовать иерархию классов, включающую абстрактный класс Device (устройство) с методами Add, GetName и GetTotalPrice и конкретные классы SimpleDevice (простое устройство) и CompoundDevice (составное устройство). Метод Add с параметром-ссылкой d типа Device добавляет устройство d в набор дочерних устройств (имеет смысл только для класса CompoundDevice; для класса SimpleDevice не выполняет никаких действий), метод GetName возвращает строковое имя данного устройства, метод GetTotalPrice возвращает стоимость данного устройства и всех его потомков (целое число). В классе Device метод Add не выполняет никаких действий, а методы GetName и GetTotalPrice являются абстрактными.

Классы SimpleDevice и CompoundDevice содержат строковое поле name и целочисленное поле price; класс CompoundDevice хранит свои дочерние устройства в виде массива или другой структуры с элементами-ссылками типа Device (можно считать, что любой объект типа CompoundDevice содержит не более 15 дочерних устройств). Конструктор классов SimpleDevice и CompoundDevice содержит два параметра: строку name и целое число price.

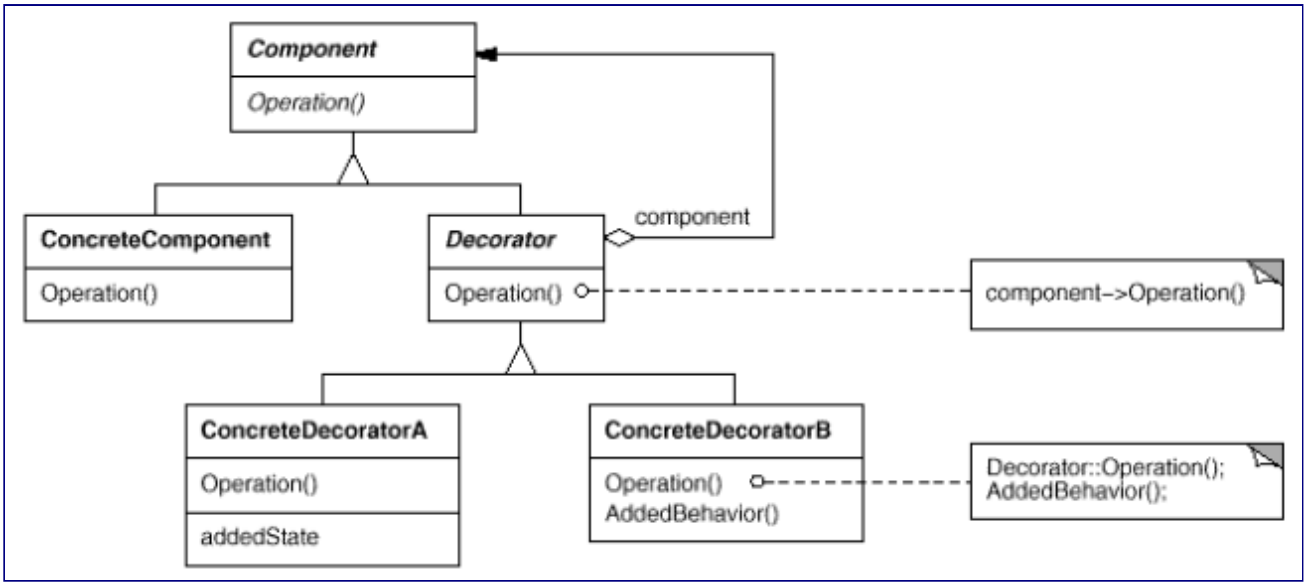
Дано целое число N (≤ 15) и N пар вида (name, price), где name — некоторая строка, а price — положительное целое число. Первый символ строки name является латинской буквой; если буква заглавная, то строка определяет составное устройство, а если строчная — то простое устройство. Кроме того, дан набор из N целых чисел, определяющих связи между исходными устройствами: число с индексом K (K = 0, …, N − 1) определяет индекс родительского устройства для исходного устройства с индексом K (при этом гарантируется, что родительское устройство обязательно имеет тип CompoundDevice). Если устройство не имеет родителя, то соответствующий элемент в исходном наборе чисел равен −1. Перебирая созданные устройства в порядке, соответствующем порядку их характеристик (name, price) и используя методы GetName и GetTotalPrice, вывести для каждого устройства название и полную стоимость.

```
public abstract class Device
{
    public virtual void Add(Device d) {}
    public abstract string GetName();
    public abstract int GetTotalPrice();
```

```
    }

    // Implement the SimpleDevice
    //   and CompoundDevice descendant classes
```

OOP2Struc7°.



Decorator (**Декоратор**) — структурный паттерн.

Известен также под именем **Wrapper** (**Обертка**).

Частота использования: средняя.

Назначение: динамически добавляет объекту новые возможности, приводящие к изменению его состояния и/или поведения. Является гибкой альтернативой порождению подклассов с целью расширения функциональности.

Участники:

• *Component* (*Компонент*) — определяет интерфейс для объектов, к которым могут быть динамически добавлены новые возможности;

• *ConcreteComponent* (*Конкретный компонент*) — определяет объект, к которому могут быть добавлены новые возможности;

• *Decorator* (*Декоратор*) — хранит ссылку на объект Component и определяет интерфейс, соответствующий интерфейсу Component;

• *ConcreteDecoratorA* и *ConcreteDecoratorB* (*Конкретные декораторы*) — добавляют к компоненту новые возможности, изменяющие его состояние и/или поведение.

Задание 1. Реализовать иерархию классов, которая включает абстрактный класс Component с абстрактным методом Show (не имеет параметров, возвращает строку), абстрактный класс Decorator, который является потомком класса Component и содержит защищенное поле comp — ссылку на объект типа Component, и конкретные классы ConcreteComponent (потомок класса Component), ConcreteDecoratorA и ConcreteDecoratorB (потомки класса Decorator).

Класс ConcreteComponent содержит строковое поле text, которое инициализируется в конструкторе с помощью одноименного параметра. Метод Show класса ConcreteComponent возвращает строку text. Классы ConcreteDecoratorA и ConcreteDecoratorB имеют конструктор с параметром comp, являющимся ссылкой на объект типа Component; этот параметр присваивается полю comp. Метод Show конкретного декоратора A возвращает строку, полученную путем добавления строки "==" перед и после текста, возвращенного методом Show объекта comp. Метод Show конкретного декоратора B возвращает строку, полученную путем добавления символа "(" перед текстом, возвращенным методом Show объекта comp, и символа ")" после этого текста. Таким образом, каждый декоратор изменяет поведение метода Show исходного объекта Component, добавляя к возвращаемому значению дополнительный префикс и суффикс.

Дано целое число N (≤ 9) и N пар строк (S, D), причем строка S является непустой, а строка D содержит только буквы "A" и "B" и может быть пустой. Создать набор из N объектов типа

Component, формируя каждый элемент этого набора на основе соответствующей пары строк (S, D) следующим образом: вначале создать объект типа ConcreteComponent, вызвав его конструктор с параметром S, а затем последовательно применять к результирующему объекту декораторы A или B, причем количество и порядок декораторов определяется строкой D (например, в случае строки "AAB" к исходному объекту типа ConcreteComponent надо последовательно применить декораторы A, A и B). Перебирая созданный набор из N объектов *в обратном порядке*, вызвать для каждого из них метод Show и вывести его возвращаемое значение.

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

// Implement the ConcreteComponent descendant class

public abstract class Decorator : Component
{
    protected Component comp;
}

// Implement the ConcreteDecoratorA
//   and ConcreteDecoratorB descendant classes
```

OOP2Struc8°. **Decorator** (**Декоратор**) — структурный паттерн.

Задание 2. Реализовать иерархию классов, включающую абстрактный класс Function с абстрактными методами GetName (без параметров; возвращает строку) и GetValue(x) (с целочисленным параметром; возвращает целое число) и пять конкретных классов — потомков Function: FX, FDouble, FTriple, FSquare, FCube.

Класс FX не имеет полей; его метод GetName возвращает строку "X", а функция GetValue(x) возвращает свой параметр x. Остальные конкретные классы являются декораторами; все они содержат ссылочное поле f типа Function и конструктор с параметром-ссылкой f типа Function, который инициализирует это поле. Метод GetName данных классов вызывает метод GetName класса f и возвращает его результат, снабженный дополнительным префиксом и суффиксом: "2*(" и ")" для FDouble, "3*(" и ")" для FTriple, "(" и ")^2" для FSquare, "(" и ")^3" для FCube. Метод GetValue(x) данных классов вызывает метод GetValue(x) класса f и возвращает его результат, преобразованный следующим образом: результат умножается на 2 для FDouble, умножается на 3 для FTriple, возводится к квадрат для FSquare, возводится в куб для FCube.

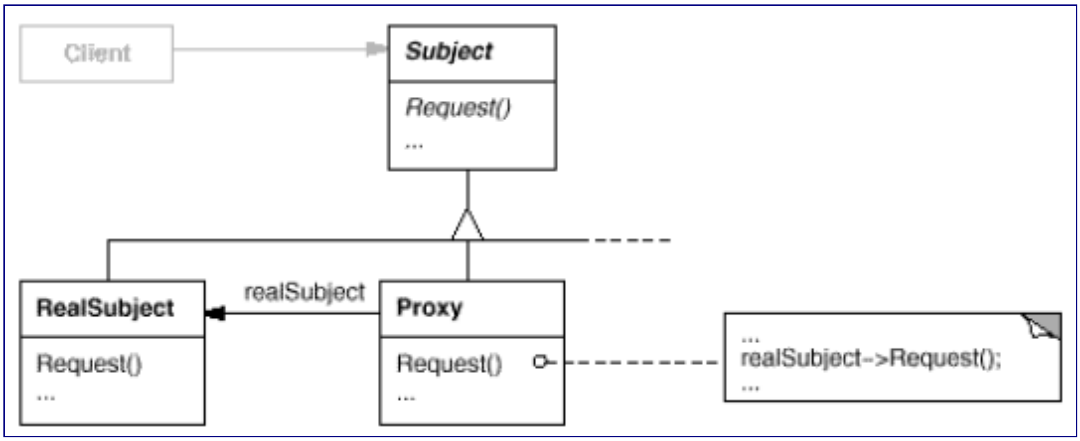
Дано целое число N (≤ 10) и набор из N строк; каждая строка содержит комбинацию из букв "D", "T", "S" и "C" и может быть пустой. Кроме того, даны два целых числа X1 и X2. Создать набор из N объектов типа Function, формируя каждый элемент следующим образом: вначале создать объект типа FX, а затем последовательно применять к результирующему объекту декораторы FDouble, FTriple, FSquare, FCube, причем количество и порядок декораторов определяется символами соответствующей строки из исходного набора (например, в случае строки "TCSS" к исходному объекту типа FX надо последовательно применить декораторы FTriple, FCube и два декоратора FSquare). Перебирая созданный набор из N объектов в исходном порядке, вызвать для каждого из них методы GetName, GetValue(X1) и GetValue(X2) и вывести их возвращаемые значения.

```
public abstract class Function
{
    public abstract string GetName();
    public abstract int GetValue(int x);
}

// Implement the FX, FDouble, FTriple, FSquare
//   and FCube descendant classes
```

Proxy, Bridge, Flyweight

OOP2Struc9°.



Proxy (**Заместитель**) — структурный паттерн.

Известен также под именем **Surrogate** (**Суррогат**).

Частота использования: выше средней.

Назначение: является суррогатом другого объекта и контролирует доступ к нему.

Участники:

• *Subject* (*Субъект*) — определяет общий для RealSubject и Proxy интерфейс, так что класс Proxy можно использовать везде, где ожидается RealSubject;

• *RealSubject* (*Реальный субъект*) — определяет реальный объект, представленный заместителем;

• *Proxy* (*Заместитель*) — хранит ссылку, которая позволяет заместителю обратиться к реальному субъекту; контролирует доступ к реальному субъекту и может отвечать за его создание и удаление; прочие обязанности зависят от вида заместителя (*удаленный заместитель* отвечает за отправление запроса реальному субъекту в другом адресном пространстве; *виртуальный заместитель* может отложить создание реального субъекта и вызывать некоторые его методы самостоятельно; *защищающий заместитель* проверяет, имеет ли вызывающий объект необходимые для выполнения запроса права).

Задание 1. Реализовать иерархию классов, включающую абстрактный класс Subject с абстрактными методами OperationA, OperationB, OperationC, OperationD и конкретные классы RealSubject и Proxy — потомки класса Subject. Указанные методы не имеют параметров и возвращают строку. Класс RealSubject не имеет полей, его методы возвращают следующие строки: "A (Real)", "B (Real)", "C (Real)", "D (Real)".

Класс Proxy является заместителем класса RealSubject, комбинирующим черты виртуального и защищающего заместителя. Предполагается, что операции A и B являются простыми и могут быть реализованы в самом заместителе, в то время как операции C и D являются сложными и доступны только в классе RealSubject. Кроме того, операции A и C являются безопасными, а операции B и D — потенциально опасными, и в некоторых ситуациях их целесообразно заблокировать. Поэтому класс Proxy содержит два логических поля: deferredMode (отложенный режим) и protectedMode (защищенный режим), которые задаются в его конструкторе и определяют его поведение по отношению к реальному субъекту. Кроме того, класс Proxy содержит поле rsubj — ссылку на объект типа RealSubject.

Если поле deferredMode равно false, то объект RealSubject создается в конструкторе класса Proxy (и связывается со ссылкой rsubj), если deferredMode равно true, то начальное значение ссылки rsubj является пустым. Если поле protectedMode равно false и при этом ссылка rsubj не является пустой, то все операции переадресуются объекту, на который ссылается rsubj. Если protectedMode равно false, а ссылка rsubj является пустой, то простые операции A и B выполняются самим объектом Proxy и при этом возвращаются строки "A (Proxy)" и "B (Proxy)", а в случае вызова операции C или D объект RealSubject создается и связывается со ссылкой rsubj, после чего эта операция переадресуется ему. Если protectedMode равно true, то выполнение операций A и C не отличается от ранее описанного, а при попытке вызова операций B и D они отменяются (независимо от значения ссылки rsubj), причем соответствующие методы возвращают строки "B denied" и "D denied".

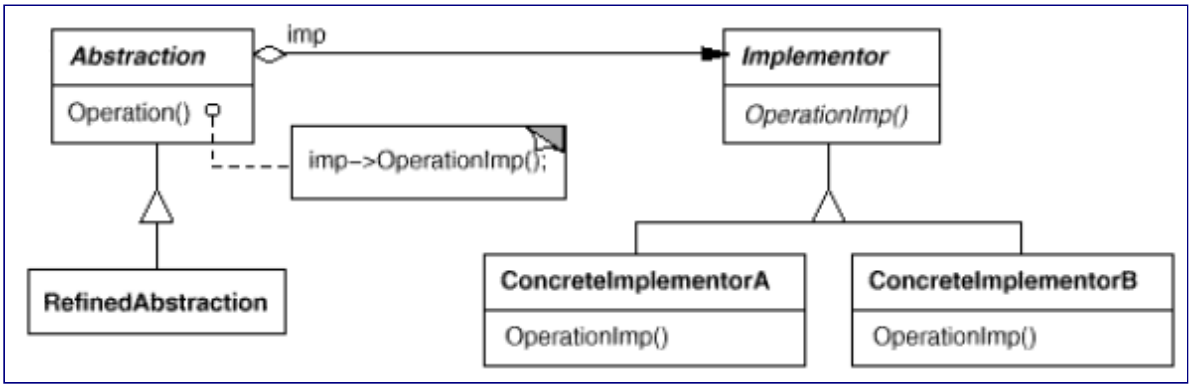
Дан набор из трех целых чисел, принимающих значения от −1 до 3, и строка, содержащая только символы из набора "A", "B", "C", "D" — имена операций. Создать массив из трех

ссылочных элементов типа Subject, инициализировав элементы конкретными объектами в зависимости от значения исходных чисел: −1 — RealSubject, 0 — Proxy(false, false), 1 — Proxy(true, false), 2 — Proxy(false, true), 3 — Proxy(true, true). Для каждого из созданных объектов выполнить набор операций, определяемый исходной строкой, и вывести результат, возвращаемый каждой операцией.

```
public abstract class Subject
{
    public abstract string OperationA();
    public abstract string OperationB();
    public abstract string OperationC();
    public abstract string OperationD();
}

// Implement the RealSubject and Proxy descendant classes
```

OOP2Struc10°.



Bridge (**Мост**) — структурный паттерн.

Известен также под именем **Handle/Body** (**Описатель/Тело**).

Частота использования: средняя.

Назначение: отделяет абстракцию от ее реализации так, чтобы то и другое можно было изменять независимо.

Участники:

• *Abstraction* (*Абстракция*) — определяет интерфейс абстракции; хранит ссылку на объект типа Implementor;

• *RefinedAbstraction* (*Уточненная абстракция*) — расширяет интерфейс, определенный абстракцией Abstraction;

• *Implementor* (*Реализатор*) — определяет интерфейс для классов реализации; не обязан точно соответствовать интерфейсу класса Abstraction (обычно предоставляет только примитивные операции, в то время как класс Abstraction определяет операции более высокого уровня, базирующиеся на этих примитивах);

• *ConcreteImplementorA* и *ConcreteImplementorB* (*Конкретные реализаторы*) — содержат конкретную реализацию интерфейса класса Implementor.

Задание 1. Реализовать иерархию классов-*реализаторов*, содержащую абстрактного реализатора Implementor и два конкретных реализатора ConcreteImplementorA и ConcreteImplementorB. Классы отвечают за представление горизонтальных линий и текста и включают методы DrawLine(size) и DrawText(text), возвращающие строковые значения. Параметр size определяет размер линии (в символах), параметр text — выводимый текст. В классе Implementor методы DrawLine и DrawText являются абстрактными. Конкретный реализатор A представляет линию в виде набора символов "−", а текст отображает в нижнем регистре. Конкретный реализатор B представляет линию в виде набора символов "=", а текст отображает в верхнем регистре. Оба конкретных реализатора имеют конструкторы без параметров, не выполняющие дополнительных действий.

Реализовать класс Abstraction, предназначенный для отображения и корректировки строки заголовка. Конструктор класса принимает параметры imp типа Implementor (ссылку на используемый реализатор) и size целого типа (размер заголовка). Класс также содержит метод Show без параметров, возвращающий строку-заголовок, и метод SetSize(n), задающий

размер заголовка равным значению n (целое неотрицательное число). Класс Abstraction реализует простейший вариант заголовка, представляющий собой линию указанного размера.

Реализовать класс RefinedAbstraction, который является усовершенствованным вариантом класса Abstraction и позволяет включать в заголовок текст. Конструктор класса RefinedAbstraction содержит, кроме параметров imp и size, имеющих тот же смысл, что и для конструктора класса Abstraction, строковый параметр caption. Заголовок формируется следующим образом: вначале указывается линия размера 1, затем строка caption, затем линия такого размера, чтобы суммарный размер заголовка был равен size. Для малых значений size строка caption может урезаться справа. Переопределить нужным образом метод Show в классе RefinedAbstraction.

Дано целое положительное число size (начальный размер заголовка) и строка caption (необязательный элемент заголовка). Также даны пять целых положительных чисел (новые размеры заголовков). Создать экземпляры классов Abstraction и RefinedAbstraction с указанными параметрами и каждым из конкретных реализаторов A и B и вывести соответствующие заголовки методом Show. Затем, используя каждый из новых размеров, изменить размер каждого заголовка и вывести измененные заголовки. Порядок вывода заголовков для каждого размера: Abstraction с реализатором A, Abstraction с реализатором B, RefinedAbstraction с реализатором A, RefinedAbstraction с реализатором B. Для хранения созданных объектов использовать массив из четырех ссылочных элементов типа Abstraction.

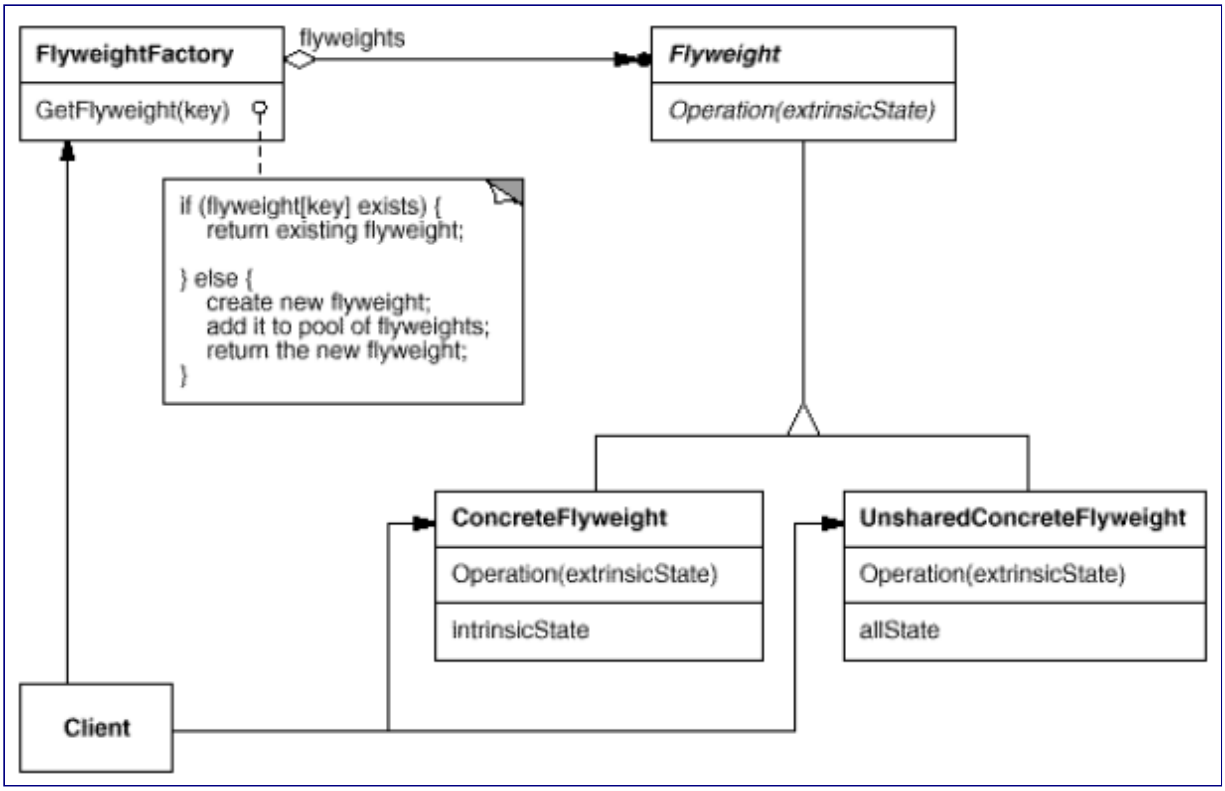
```
public abstract class Implementor
{
    public abstract string DrawLine(int size);
    public abstract string DrawText(string text);
}

// Implement the ConcreteImplementorA
//   and ConcreteImplementorB descendant classes

public class Abstraction
{
    protected int size;
    protected Implementor imp;
    public Abstraction(Implementor imp, int size)
    {
        this.imp = imp;
        this.size = size;
    }
    // Complete the implementation of the class
}

// Implement the RefinedAbstraction descendant class
```

OOP2Struc11°.



Flyweight (**Приспособленец**) — структурный паттерн.

Частота использования: низкая.

Назначение: применяет *разделение* (т. е. совместное использование одного и того же экземпляра) для эффективной поддержки множества объектов, сохраняя основную часть их состояния во внешних данных клиента.

Участники:

• *Flyweight* (*Приспособленец*) — объявляет интерфейс, с помощью которого приспособленцы могут получать внешнее состояние или воздействовать на него;

• *ConcreteFlyweight* (*Конкретный приспособленец*) — реализует интерфейс класса Flyweight и добавляет при необходимости внутреннее состояние (объект класса ConcreteFlyweight должен быть разделяемым, любое сохраняемое им состояние должно быть *внутренним*, то есть не зависящим от контекста);

• *UnsharedConcreteFlyweight* (*Неразделяемый конкретный приспособленец*) — реализует интерфейс класса Flyweight, но не является разделяемым; данный класс может сохранять состояние, зависящее от контекста;

• *FlyweightFactory* (*Фабрика приспособленцев*) — создает объекты-приспособленцы и управляет ими; при запросе клиентом приспособленца объект FlyweightFactory предоставляет существующий экземпляр или создает новый, если готового еще нет;

• *Client* (*Клиент*) — хранит ссылки на одного или нескольких приспособленцев; вычисляет или хранит внешнее состояние приспособленцев.

Задание 1. Реализовать иерархию классов, которая включает абстрактный класс Flyweight с абстрактным методом Show(state), имеющим логический параметр state и возвращающим символьное значение, и конкретные классы ConcreteFlyweight и UnsharedConcreteFlyweight — потомки класса Flyweight. Предполагается, что клиент будет использовать большое количество объектов класса ConcreteFlyweight, поэтому этот класс должен поддерживать разделение. Метод Show(state) класса ConcreteFlyweight возвращает символ "A", регистр которого зависит от значения state (если state = true, то используется верхний регистр, если state = false, то нижний). Конструктор класса ConcreteFlyweight не имеет параметров и не выполняет дополнительных действий.

Класс UnsharedConcreteFlyweight не является разделяемым; он хранит дополнительное символьное поле inf, которое инициализируется в конструкторе, имеющем соответствующий символьный параметр. Метод Show(state) класса UnsharedConcreteFlyweight возвращает

символ inf, его регистр определяется параметром state (как в методе Show класса ConcreteFlyweight).

Реализовать класс-фабрику FlyweightFactory, содержащий поле-ссылку cf типа ConcreteFlyweight и поле count целого типа, а также конструктор без параметров (при создании данного объекта поле cf содержит пустую ссылку, а поле count — значение 0). Класс содержит метод CreateFlyweight(inf) с символьным параметром и возвращаемым значением-ссылкой типа Flyweight. Если параметр отличен от символа "A" или "a", то фабрика создает и возвращает новый объект типа UnsharedConcreteFlyweight, используя конструктор с параметром inf. Если параметр представляет собой символ "A" или "a", то в случае, если поле cf содержит ссылку на существующий объект типа ConcreteFlyweight, возвращается этот объект, если же поле cf является пустым, то объект типа ConcreteFlyweight создается, сохраняется в поле cf и возвращается методом CreateFlyweight. Если в методе CreateFlyweight создается новый объект, то поле count увеличивается на 1; таким образом, данное поле хранит общее количество созданных объектов. Значение поля count можно получить с помощью метода GetCount без параметров.

Реализовать также класс Client, предназначенный для создания, хранения и обработки наборов объектов типа Flyweight. Класс содержит поля f — объект типа FlyweightFactory и fw — структуру ссылочных данных (например, массив) с элементами типа Flyweight (можно считать, что число элементов fw не превосходит 30). Конструктор класса Client не имеет параметров, в нем создается объект f и инициализируется структура данных fw. Класс содержит методы MakeFlyweights(inf), ShowFlyweights(state) и GetFlyweightCount. Параметр inf метода MakeFlyweights является строкой, определяющей набор создаваемых объектов типа Flyweight (для создания требуемого набора в методе MakeFlyweights в цикле вызывается метод CreateFlyweight объекта f с параметром — очередным символом строки inf). Ссылки на созданные объекты сохраняются в структуре fw; при каждом вызове метода MakeFlyweights прежнее содержимое структуры fw очищается. Метод ShowFlyweights(state) возвращает строку, состоящую из символов, возвращаемых методом Show(state) для каждого элемента набора fw. Метод GetFlyweightCount без параметров возвращает количество объектов, созданных к данному моменту фабрикой f; для этого используется ее метод GetCount.

Дано пять текстовых строк; длина каждой строки не превосходит 30, большинство символов в этих строках являются символами "A" в верхнем или нижнем регистре. Создать объект типа Client и для каждой исходной строки s вызвать метод MakeFlyweights(s) и вывести значения, возвращаемые методами ShowFlyweights(true), ShowFlyweights(false) и GetFlyweightCount.

```
public abstract class Flyweight
{
    public abstract char Show(bool state);
}

// Implement the ConcreteFlyweight
//   and UnsharedConcreteFlyweight descendant classes

// Implement the FlyweightFactory and Client classes
```

Page generation date: 2022/4/26.