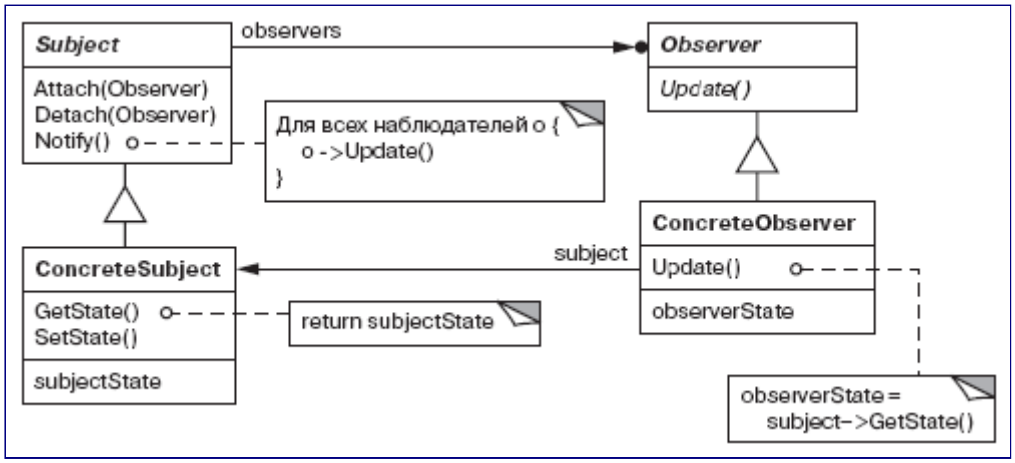
Паттерны поведения

М. Э. Абрамян, 2021

Observer, Strategy, Template Method

OOP3Behav1°.



Observer (**Наблюдатель**) — паттерн поведения.

Известен также под именем **Publish-Subscribe** (**Издатель-Подписчик**).

Частота использования: высокая.

Назначение: определяет зависимость типа "один-ко-многим" между объектами таким образом, что при изменении состояния одного объекта все зависящие от него объекты-наблюдатели оповещаются об этом событии.

Участники:

• *Subject* (*Субъект*) — располагает информацией о своих наблюдателях (за субъектом может следить любое число наблюдателей); предоставляет интерфейс для присоединения и отсоединения наблюдателей;

• *Observer* (*Наблюдатель*) — определяет интерфейс обновления для объектов, которые должны быть уведомлены об изменении субъекта;

• *ConcreteSubject* (*Конкретный субъект*) — сохраняет состояние, представляющее интерес для конкретного наблюдателя ConcreteObserver; посылает информацию своим наблюдателям, когда происходит изменение;

• *ConcreteObserver* (*Конкретный наблюдатель*) — хранит ссылку на объект класса ConcreteSubject; сохраняет данные, которые должны быть согласованы с данными субъекта.

Задание 1. Реализовать две иерархии классов. Первая иерархия включает абстрактный класс Subject с абстрактными методами Attach(observ), Detach(observ) и Notify и класс-потомок ConcreteSubject. Методы Attach и Detach имеют параметр-ссылку типа Observer и не возвращают значений. Метод Notify не имеет параметров и не возвращает значения.

В классе ConcreteSubject дополнительно описать поля observers и state. Поле observers является структурой с элементами-ссылками типа Observer, в которой хранятся все наблюдатели, присоединенные в настоящий момент к субъекту (можно считать, что наблюдателей не более 10). В качестве такой структуры удобно использовать динамическую структуру, имеющую команды для добавления и удаления элементов. Метод Attach(observ) добавляет объект observ в структуру observers, метод Detach(observ) удаляет объект observ из данной структуры. Поле state имеет символьный тип и моделирует текущее состояние субъекта. С ним связаны два метода: метод SetState(st) изменяет поле state, присваивая ему значение символьного параметра st и дополнительно вызывает метод Notify; метод GetState (без параметров) возвращает текущее значение поля state. В методе Notify выполняется перебор элементов структуры observers: для каждого элемента структуры вызывается его метод Update. При реализации метода Notify необходимо учесть ситуацию, когда во время выполнения метода Update некоторые наблюдатели отсоединятся от субъекта, что приведет к изменению структуры observers. Конструктор класса ConcreteSubject не имеет параметров, поле state инициализируется символом "пробел".

Вторая иерархия включает абстрактный класс Observer с абстрактным методом Update (не имеет параметров и не возвращает значения) и конкретный класс ConcreteObserver, имеющий строковое поле log, символьное поле detachInfo, поле subj — ссылку на объект типа ConcreteSubject, а также методы Attach, Detach и GetLog. Конструктор класса ConcreteObserver имеет параметры subj и detachInfo, которые инициализируют соответствующие поля; поле log инициализируется пустой строкой. Методы Attach и Detach не имеют параметров и не возвращают значения. В методе Attach выполняется вызов метода Attach объекта subj, в методе Detach выполняется вызов метода Detach объекта subj, причем в качестве параметра в обоих вызываемых методах передается ссылка на объект ConcreteObserver, из которого вызваны эти методы. Метод GetLog не имеет параметров и возвращает значение поля log.

Метод Update, переопределяемый в классе ConcreteObserver, является основным методом, обеспечивающим взаимодействие между конкретным субъектом и конкретным наблюдателем. Именно этот метод вызывается конкретным субъектом при изменении его состояния. Таким образом, его вызов означает, что состояние конкретного субъекта изменилось, и наблюдатель может узнать это новое состояние, вызвав в методе Update метод GetState объекта subj. В данном задании в методе Update надо выполнить следующие дополнительные действия: добавить символ, полученный методом GetState, в конец поля log объекта ConcreteObserver, для которого был вызван метод Update, и, кроме того, если полученный символ совпадает со значением поля detachInfo, то немедленно отсоединить наблюдателя ConcreteObserver от субъекта subj, вызвав метод Detach наблюдателя.

Дано целое число N (≤ 10) и строка S, содержащая различные заглавные латинские буквы и имеющая длину не более 26. Создать объект subj типа ConcreteSubject и набор observers из N объектов типа ConcreteObserver, указав в качестве первого параметра конструктора объектов ConcreteObserver ссылку на объект subj, а в качестве второго параметра — заглавные латинские буквы, перебирая их в алфавитном порядке ("A" для первого объекта ConcreteObserver, "B" для второго объекта и т. д.). Для каждого объекта ConcreteObserver вызвать его метод Attach. Затем для каждого символа из данной строки S вызвать метод SetState объекта subj, передав методу этот символ в качестве параметра. После обработки всех символов строки S вывести значения полей log объектов из набора observers, используя метод GetLog класса ConcreteObserver (выведенные строки должны содержать все начальные символы строки S вплоть до того символа, который вызвал отсоединение наблюдателя от объекта subj).

Примечание. В языке C# для реализации паттерна Observer удобно использовать *делегаты* и *события*. В нашем случае можно удалить методы Attach и Detach из иерархии классов Subject (оставив в абстрактном классе только метод Notify), удалить структуру observers из класса ConcreteSubject, описать делегат: `public delegate void NotifyEventHandler();` добавить событие OnNotify в класс ConcreteSubject: `public event NotifyEventHandler OnNotify;` и определить метод Notify следующим образом: `if (OnNotify != null) OnNotify();` В классе ConcreteObserver методы Attach и Detach будут содержать единственный оператор: `subj.OnNotify += Update` и `subj.OnNotify -= Update` соответственно.

```
public abstract class Subject
{
    public abstract void Attach(Observer observ);
    public abstract void Detach(Observer observ);
    public abstract void Notify();
}

// Implement the ConcreteSubject descendant class

public abstract class Observer
{
    public abstract void Update();
}

// Implement the ConcreteObserver descendant class
```

OOP3Behav2°. **Observer (Наблюдатель)** — паттерн поведения.

Задание 2. Реализовать вариант взаимодействия субъектов и наблюдателей, не требующий специальных методов доступа к состоянию субъекта и упрощающий взаимодействие наблюдателя с *несколькими* субъектами.

Первая иерархия классов включает абстрактный класс Subject с абстрактными методами Attach(observ), Detach(observ), SetInfo(info) и класс-потомок ConcreteSubject. Методы Attach и Detach имеют параметр-ссылку типа Observer и не возвращают значений. Метод SetInfo имеет строковый параметр и также не возвращает значения. В классе ConcreteSubject дополнительно описать поле observers — структуру с элементами-ссылками типа Observer, в которой хранятся все наблюдатели, присоединенные в настоящий момент к субъекту (можно считать, что наблюдателей не более 10). В качестве такой структуры удобно использовать динамическую структуру, имеющую команды для добавления и удаления элементов. Метод Attach(observ) добавляет объект observ в структуру observers, метод Detach(observ) удаляет объект observ из данной структуры. В методе SetInfo(info) выполняется перебор элементов структуры observers и для каждого элемента этой структуры вызывается его метод OnInfo(sender, info), причем в качестве параметра info указывается параметр метода SetInfo, а в качестве параметра sender — ссылка на объект ConcreteSubject, вызвавший метод SetInfo. Таким образом, наблюдатель сразу информируется и о наступившем событии, и о субъекте, который инициировал это событие. При реализации метода SetInfo необходимо учесть ситуацию, когда некоторые наблюдатели в своем методе OnInfo отсоединятся от субъекта, что приведет к изменению структуры observers.

Вторая иерархия включает абстрактный класс Observer с абстрактным методом OnInfo(sender, info) и класс-потомок ConcreteObserver. Параметр sender метода OnInfo является ссылкой на объект типа Subject, параметр info — строковый. Класс ConcreteObserver дополнительно имеет строковое поле log и символьное поле detachInfo, а также методы Attach, Detach и GetLog. Конструктор класса ConcreteObserver имеет параметр detachInfo, который инициализирует соответствующее поле; поле log инициализируется пустой строкой. Методы Attach(subj) и Detach(subj) имеют параметр-ссылку типа Subject и не возвращают значений. В методе Attach выполняется вызов метода Attach объекта subj, в методе Detach выполняется вызов метода Detach объекта subj, причем в качестве параметра в обоих вызываемых методах передается ссылка на объект ConcreteObserver, из которого вызваны эти методы. Метод GetLog не имеет параметров и возвращает значение поля log.

Метод OnInfo(sender, info), переопределяемый в классе ConcreteObserver, является основным методом, обеспечивающим взаимодействие между конкретным субъектом и конкретным наблюдателем. Напомним, что этот метод вызывается конкретным субъектом для информирования всех присоединенных к нему в настоящий момент наблюдателей, причем информация передается в поле info, а поле sender содержит ссылку на субъект, передавший эту информацию. В данном задании в методе OnInfo надо выполнить следующие действия: добавить содержимое параметра info в конец строки log объекта ConcreteObserver, для которого был вызван метод OnInfo, и, кроме того, если *последний* символ строки info совпадает со значением поля detachInfo объекта ConcreteObserver, то необходимо отсоединить этого наблюдателя от субъекта sender, вызвав метод Detach объекта ConcreteObserver с параметром sender.

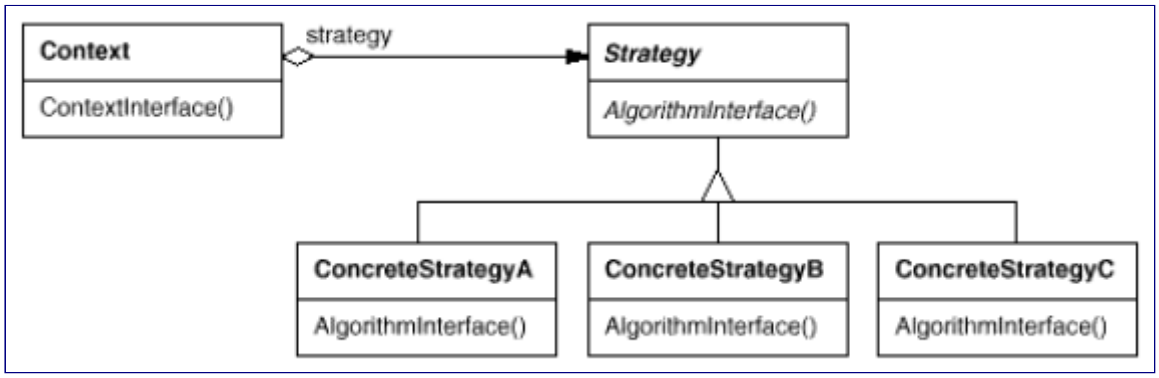
Дано целое число N (≤ 10). Кроме того, дано целое число K (≤ 30) и набор различных двухсимвольных строк, первым символом которых является цифра "1" или "2", а вторым — строчная латинская буква. Создать два объекта subj1 и subj2 типа ConcreteSubject и набор observers из N объектов типа ConcreteObserver, указывая в качестве параметра detachInfo конструктора объектов ConcreteObserver строчные латинские буквы, которые перебираются в алфавитном порядке ("a" для первого объекта ConcreteObserver, "b" для второго объекта и т. д.). Для каждого объекта ConcreteObserver вызвать методы Attach с параметрами-ссылками на объекты subj1 и subj2. Затем для каждой строки из данного набора строк вызвать метод SetInfo SetInfo объекта subj1 или subj2, передав эту строку в качестве параметра, причем если строка начинается с цифры "1", то метод SetInfo должен вызываться для объекта subj1, а если с цифры "2", то для объекта subj2. После обработки всех строк из исходного набора вывести значения полей log объектов из набора observ, используя метод GetLog класса ConcreteObserver (выведенные строки должны содержать

все данные, переданные наблюдателям субъектами subj1 и subj2, вплоть до тех данных, которые вызвали отсоединение наблюдателя от соответствующего субъекта).

Примечание. Описанный вариант взаимодействия субъектов и наблюдателей можно реализовать с помощью *делегаты* и *события* (см. примечание к заданию OOP3Behav1).

```
public abstract class Subject
{
    public abstract void Attach(Observer observ);
    public abstract void Detach(Observer observ);
    public abstract void SetInfo(string info);
}

// Implement the ConcreteSubject descendant class

public abstract class Observer
{
    public abstract void OnInfo(Subject sender, string info);
}

// Implement the ConcreteObserver descendant class
```

OOP3Behav3°.



Strategy (**Стратегия**) — паттерн поведения.

Известен также под именем **Policy** (**Политика**).

Частота использования: выше средней.

Назначение: определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет заменять алгоритмы для любого клиента ("контекста") на этапе выполнения.

Участники:

• *Strategy* (*Стратегия*) — объявляет общий для всех поддерживаемых алгоритмов интерфейс; класс Context пользуется этим интерфейсом для вызова конкретного алгоритма, определенного в одном из классов ConcreteStrategy;

• *ConcreteStrategyA*, *ConcreteStrategyB*, *ConcreteStrategyC* (*Конкретные стратегии*) — реализуют алгоритм, используя интерфейс, объявленный в классе Strategy;

• *Context* (*Контекст*) — конфигурируется объектом класса ConcreteStrategy; хранит ссылку на объект класса Strategy; может определять интерфейс, который позволяет объекту Strategy получить доступ к данным контекста.

Задание 1. Реализовать две иерархии классов. Первая иерархия включает абстрактный класс Strategy с абстрактным методом Algorithm(info) (метод имеет строковый параметр info и возвращает строковое значение) и конкретные классы ConcreteStrategyA, ConcreteStrategyB и ConcreteStrategyC — потомки класса Strategy. Конкретные классы имеют конструкторы без параметров, не выполняющие дополнительных действий. Метод Algorithm(info) конкретных классов A, B, C возвращает строку, полученную из строки info добавлением справа символа "A", "B", "C" соответственно.

Вторая иерархия включает абстрактный класс Context и конкретные классы Context1 и Context2 — потомки класса Context. Класс Context содержит поле st — ссылку на объект типа Strategy — и методы SetStrategy(st) (с параметром-ссылкой st типа Strategy) и RunAlgorithm (без параметров, возвращает строковое значение). Метод SetStrategy класса

Context присваивает свой параметр полю st, метод RunAlgorithm является абстрактным. Классы Context1 и Context2 переопределяют метод RunAlgorithm, вызывая в нем метод Algorithm для поля st, причем в качестве параметра info указывается строка "1" (для класса Context1) или "2" (для класса Context2); метод RunAlgorithm возвращает строку, возвращаемую методом Algorithm. Конструкторы классов Context1 и Context2 не имеют параметров; в них создается объект типа ConcreteStrategyA (для класса Context1) или ConcreteStrategyB (для класса Context2) и поле st инициализируется ссылкой на созданный объект.

Дана строка S, состоящая из символов "1" и "2"; ее длина не превосходит 15. Также дано целое число K, не превосходящее длины строки S, и набор ind из K различных целых чисел, определяющих индексы символов в строке S (символы индексируются от 0). Создать набор ctxt объектов-ссылок типа Context (можно использовать массив или другую структуру данных); размер набора равен длине строки S, фактический тип каждого элемента набора определяется соответствующим символом строки S: это Context1 для символа "1" и Context2 для символа "2". Для элементов набора ctxt, индексы которых входят в исходный набор ind, вызвать метод SetStrategy, передав в качестве параметра объект типа ConcreteStrategyC. Перебирая все объекты набора ctxt в исходном порядке, вызвать для них метод RunAlgorithm и вывести его возвращаемое значение.

```
public abstract class Strategy
{
    public abstract string Algorithm(string info);
}

// Implement the ConcreteStrategyA, ConcreteStrategyB
//   and ConcreteStrategyC descendant classes

public abstract class Context
{
    protected Strategy st;
    public void SetStrategy(Strategy st)
    {
        this.st = st;
    }
    public abstract string RunAlgorithm();
}

// Implement the Context1 and Context2 descendant classes
```

OOP3Behav4°. **Strategy** (**Стратегия**) — паттерн поведения.

Задание 2. Реализовать иерархию классов-*валидаторов*, включающую класс Validator и его классы-потомки EmptyValidator, NumberValidator и RangeValidator. Классы-валидаторы предназначены для проверки правильности введенных строковых данных. Классы Validator, EmptyValidator и NumberValidator не имеют полей, их конструкторы не имеют параметров и не выполняют дополнительных действий. В классе Validator определен метод Validate(s), имеющий строковый параметр s и возвращающий строку с описанием ошибки, обнаруженной в строке s. Метод Validate класса Validator всегда возвращает пустую строку; таким образом, класс Validator считает допустимой любую строку.

Для класса EmptyValidator метод Validate(s) возвращает пустую строку, если параметр s не является пустой строкой; в противном случае он возвращает строку "!Empty text".

Для класса NumberValidator метод Validate(s) возвращает пустую строку, если параметр s содержит строковое представление некоторого целого числа; в противном случае он возвращает строку вида "!'<s>': not a number", где в позиции <s> указывается содержимое параметра s.

Класс RangeValidator содержит целочисленные поля min и max; его конструктор имеет целочисленные параметры a и b, инициализирующие поля таким образом, чтобы поле min было равно минимальному из чисел a и b, а поле max — максимальному из этих чисел. Для класса RangeValidator метод Validate(s) возвращает пустую строку, если параметр s содержит строковое представление некоторого целого числа и при этом данное число лежит в диапазоне от min до max включительно; в противном случае метод возвращает строку

вида "!'<s>': not in range <min>..<max>", где в позиции <s> указывается содержимое параметра s, а в позициях <min> и <max> — значения соответствующих полей.

Реализовать класс TextBox, содержащий строковое поле text и поле v — ссылку на объект типа Validator. Конструктор класса не имеет параметров, в нем создается объект типа Validator и ссылка на него присваивается полю v, а поле text инициализируется пустой строкой. Класс TextBox включает три метода: SetText(text) — задает или изменяет поле text; SetValidator(v) — изменяет поле v; Validate (без параметров) — вызывает для объекта v метод Validate с параметром text и возвращает значение, возвращенное этим методом.

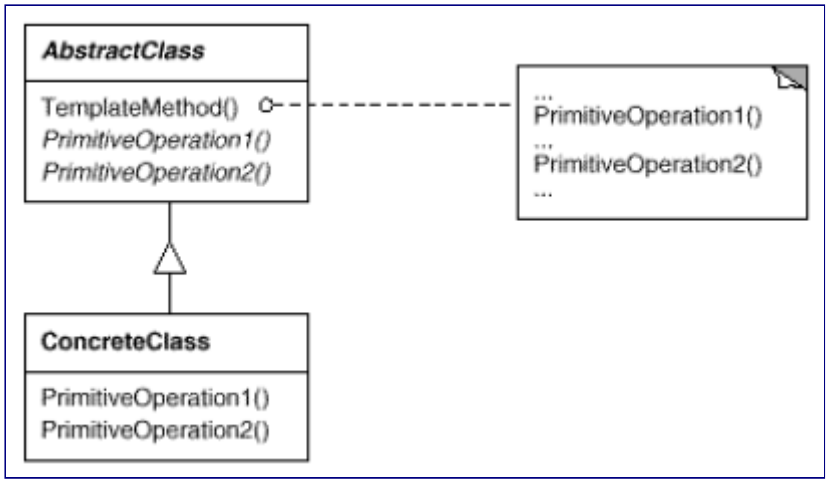
Также реализовать класс TextForm. Он содержит набор tb элементов типа TextBox (можно использовать массив или другую структуру данных). Конструктор класса TextForm имеет параметр n, определяющий размер набора tb (можно считать, что параметр n не превосходит 10); в конструкторе создаются все элементы набора tb. Класс TextForm включает три метода: SetText(ind, text) — задает или изменяет поле text для элемента набора tb с индексом ind; SetValidator(ind, v) — изменяет поле v для элемента набора tb с индексом ind; Validate (без параметров) — последовательно вызывает методы Validate для всех элементов набора tb и возвращает строку, полученную объединением строк, возвращенных этими методами. При реализации методов SetText и SetValidator можно считать, что параметр ind всегда лежит в допустимом диапазоне (от 0 до n − 1, где n — размер набора tb).

Даны три целых числа N, A, B, причем N лежит в диапазоне от 1 до 10. Также дано целое число K, не превосходящее N, и набор из K пар (ind, val), где ind является целым числом в диапазоне от 0 до N − 1, а val является одним из символов "E", "N", "R". Все значения ind являются различными. Кроме того, дано пять наборов строк, каждый из которых содержит по N элементов.

Создать объект tf типа TextForm, вызвав его конструктор с параметром N. Для каждой пары (ind, val) вызвать метод SetValidator объекта tf с первым параметром ind и вторым параметром — ссылкой на объект-валидатор, тип которого соответствует символу val: "E" — EmptyValidator, "N" — NumberValidator, "R" — RangeValidator; для объекта RangeValidator использовать конструктор с параметрами A и B, где A и B — ранее указанные числа. Для каждого из пяти данных наборов строк выполнить следующие действия: добавить набор строк в объект tf (вызвав требуемое число раз метод SetText объекта tf) и проверить правильность этого набора строк (вызвав метод Validate объекта tf и выведя его результат).

```
public class Validator
{
    public virtual string Validate(string s)
    {
        return "";
    }
}

// Implement the EmptyValidator, NumberValidator
//   and RangeValidator descendant classes

// Implement the TextBox and TextForm classes
```

OOP3Behav5°.



Template Method (**Шаблонный метод**) — паттерн поведения.

Частота использования: средняя.

Назначение: определяет основу алгоритма и позволяет подклассам определить (или переопределить) некоторые шаги алгоритма, не изменяя его структуру в целом.

Участники:

• *AbstractClass* (*Абстрактный класс*) — определяет абстрактные примитивные операции, замещаемые в конкретных подклассах для реализации шагов алгоритма; реализует шаблонный метод, определяющий последовательность действий алгоритма (шаблонный метод вызывает примитивные операции, а также операции, определенные в классе AbstractClass или в других объектах);

• *ConcreteClass* (*Конкретный класс*) — реализует примитивные операции, выполняющие шаги алгоритма способом, который зависит от подкласса.

Помимо абстрактных примитивных операций шаблонного метода, которые *необходимо* переопределить в конкретных классах, шаблонный метод может включать операции-*перехватчики* (hooks), которые могут быть переопределены в конкретных класса, а могут быть оставлены без изменения. Использование шаблонного метода позволяет избавиться от избыточного кода в подклассах.

Задание 1. Реализовать иерархию классов, связанную с формированием рецептов приготовления напитков и включающую абстрактный класс AbstractClass и четыре его потомка: ConcreteClass1, ConcreteClass2 (непосредственные потомки абстрактного класса), ConcreteClass3 (потомок ConcreteClass1) и ConcreteClass4 (потомок ConcreteClass2). В абстрактном классе реализовать шаблонный метод TemplateMethod, формирующий и возвращающий строковое значение. Это значение получается путем последовательного добавления к результирующей строке значений, возвращаемых методами BasicOperation1, PrimitiveOperation, BasicOperation2 и HookOperation. Метод PrimitiveOperation является абстрактным методом, остальные методы имеют реализацию: метод BasicOperation1 возвращает строку "Boil water" (вскипятить воду), метод BasicOperation2 возвращает строку "=Pour into a cup" (налить в чашку), метод HookOperation возвращает пустую строку, причем данный метод является защищенным.

В классе ConcreteClass1 реализовать метод PrimitiveOperation, возвращающий строку "=Brew tea" (заварить чай), в классе ConcreteClass2 реализовать этот же метод, возвращающий строку "=Brew coffee" (заварить кофе). В классе ConcreteClass3 переопределить метод HookOperation таким образом, чтобы он возвращал строку "=Add sugar and lemon" (добавить сахар и лимон), в классе ConcreteClass4 переопределить этот же метод так, чтобы он возвращал строку "=Add sugar and milk" (добавить сахар и молоко). Конструкторы всех конкретных классов не имеют параметров и не выполняют дополнительных действий.

Дано целое число N (≤ 10) и набор из N целых чисел, принимающих значения от 1 до 4. Создать структуру (например, массив) из N элементов-ссылок типа AbstractClass и инициализировать ее элементы экземплярами конкретных классов в зависимости от значений соответствующих чисел исходного набора (если число равно 1, то создается экземпляр класса ConcreteClass1, если число равно 2, то создается экземпляр класса

ConcreteClass2 и т. д.). Перебирая элементы созданной структуры *в обратном порядке*, вызвать для каждого из них метод TemplateMethod и вывести возвращенную им строку.

```
public abstract class AbstractClass
{
    public abstract string PrimitiveOperation();
    // Implement methods TemplateMethod,
    // BasicOperation1, BasicOperation2 and HookOperation
}

// Implement the ConcreteClass1, ConcreteClass2, ConcreteClass3
//   and ConcreteClass4 descendant classes
```

OOP3Behav6°. **Template Method (Шаблонный метод)** — паттерн поведения.

Задание 2. Реализовать иерархию классов, связанную с поиском минимальных и максимальных элементов в наборе данных и включающую абстрактный класс AbstractComparable и три его потомка: NumberComparable, LengthComparable и TextComparable. В абстрактном классе реализованы четыре *статических* шаблонных метода IndexMax, LastIndexMax, IndexMin, LastIndexMin. Параметром каждого метода является набор comp объектов-ссылок типа AbstractComparable; (массив или другая структура данных); методы возвращают целое число — индекс первого наибольшего, последнего наибольшего, первого наименьшего и последнего наименьшего элемента набора comp соответственно (индексирование производится от нуля). В шаблонных методах используется абстрактный метод CompareTo(other) с параметром-ссылкой other типа AbstractComparable. Этот метод позволяет сравнивать между собой экземпляры a и b одного и того же класса — потомка класса AbstractComparable: вызов Comparable(b) для объекта a возвращает отрицательное значение, если a "меньше", чем b; нулевое значение, если a "равно" b; положительное значение, если a "больше", чем b (слова "меньше", "равно" и "больше" взяты в кавычки, так как смысл сравнений может быть различным для разных потомков класса AbstractComparable).

Каждый из конкретных классов имеет конструктор со строковым параметром data, который определяет значение поля key. Для класса NumberComparable поле key имеет целый тип; если параметр data является строковым представлением некоторого целого числа, то в поле key записывается это число, если параметр data не удовлетворяет указанному условию, то поле key полагается равным 0. Для класса LengthComparable поле key также имеет целый тип и полагается равным длине строки data. Для класса TextComparable поле key имеет строковый тип и полагается равным самой строке data.

Реализовать в классах-потомках метод CompareTo(other) в котором параметр other преобразуется к типу данного класса-потомка, после чего поля key объекта, вызвавшего метод, и объекта other сравниваются между собой; если поле key объекта, вызвавшего метод, меньше, равно или больше поля key объекта other, то метод возвращает соответственно отрицательное, нулевое или положительное значение (для класса TextComparable строковые поля key сравниваются лексикографически). Не требуется особым образом обрабатывать ситуацию, когда параметр other не может быть преобразован к нужному типу, поскольку она не будет возникать при обработке правильно сформированных наборов данных.

Даны целые числа N (≤ 9), K (≤ 7), а также K наборов из N + 1 строки, причем начальная строка в каждом наборе имеет вид "N", "L" или "T". Описать структуру comp из N элементов-ссылок типа AbstractComparable (тип структуры должен совпадать с типом параметра шаблонных методов) и использовать эту структуру для обработки *каждого* исходного набора строк следующим образом:

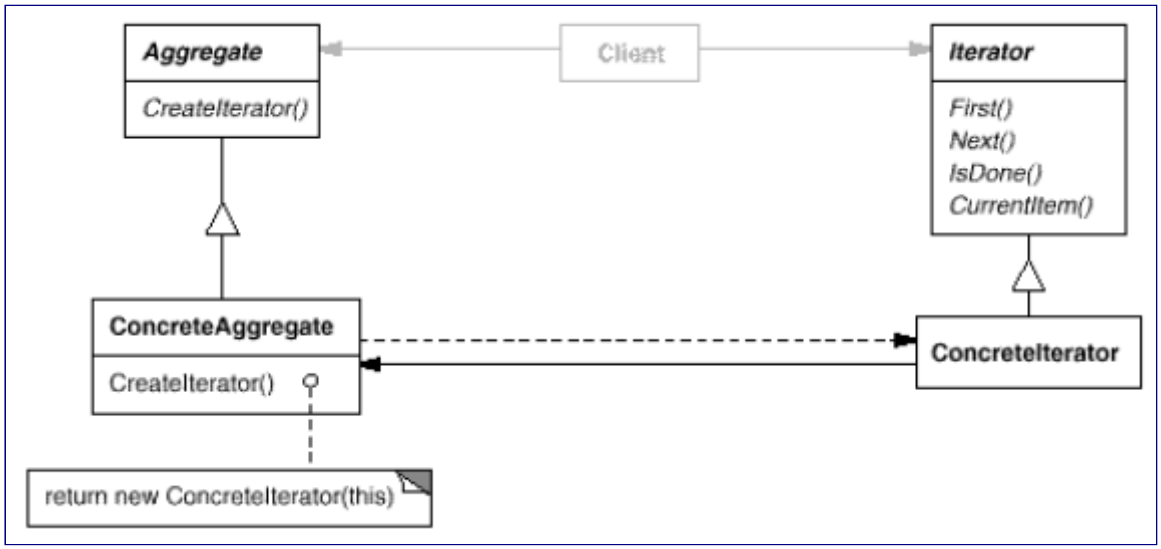
(1) создать и записать в структуру comp объекты типа, определяемого начальной строкой обрабатываемого набора: если начальная строка равна "N", "L" или "T", то создаются объекты типа NumberComparable, LengthComparable или TextComparable соответственно; строки исходного набора указываются в качестве параметров конструкторов объектов, для начальной строки объект не создается;

(2) вызвать шаблонные методы IndexMax, LastIndexMax, IndexMin, LastIndexMin с параметром comp и вывести их возвращаемые значения в указанном порядке.

```
public abstract class AbstractComparable
{
    public abstract int CompareTo(AbstractComparable other);

    // Implement the IndexMax, LastIndexMax, IndexMin
    //   and LastIndexMin static methods
}

// Implement the NumberComparable, LengthComparable
//   and TextComparable descendant classes
```

Iterator, Command, State

OOP3Behav7°.



Iterator (**Итератор**) — паттерн поведения.

Известен также под именем **Cursor** (**Курсор**).

Частота использования: высокая.

Назначение: предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления.

Участники:

• *Iterator* (*Итератор*) — определяет интерфейс для доступа и обхода элементов;

• *ConcreteIterator* (*Конкретный итератор*) — реализует интерфейс класса Iterator; следит за текущей позицией при обходе агрегата;

• *Aggregate* (*Агрегат*) — определяет интерфейс для создания объекта-итератора;

• *ConcreteAggregate* (*Конкретный агрегат*) — реализует интерфейс создания итератора и возвращает экземпляр подходящего класса ConcreteIterator.

Во многих современных языках программирования итераторы реализованы в стандартных библиотеках или даже на уровне языковых конструкций. Данное задание можно выполнять, используя либо базовые средства ООП, либо специализированные средства выбранного языка.

Задание 1. Реализовать две иерархии классов, связанные с применением паттерна Iterator. Первая иерархия является иерархией классов-*агрегатов* и включает абстрактный класс Aggregate, содержащий абстрактный метод CreateIterator (не имеет параметров, возвращает ссылку на объект Iterator), и классы ConcreteAggregateA, ConcreteAggregateB и ConcreteAggregateC. Каждый из конкретных классов содержит поле data; для класса ConcreteAggregateA оно целочисленное, для класса ConcreteAggregateB оно строковое, для класса ConcreteAggregateC оно представляет собой структуру с целочисленными элементами (например, массив; можно считать, что число элементов структуры не превосходит 10). Поле data инициализируется в конструкторе класса с помощью параметра data того же типа, что и инициализируемое поле. Конкретные классы-агрегаты A, B и C реализуют метод CreateIterator возвращающий итератор, тип которого определяется типом класса-агрегата: для агрегата A это ConcreteIteratorA, для агрегата B — ConcreteIteratorB,

для агрегата C — ConcreteIteratorC. При создании итератора в методе CreateIterator конструктору итератора передается параметр, являющийся ссылкой на объект-агрегат, вызвавший метод CreateIterator. Классы-агрегаты также имеют метод GetData без параметров, возвращающий поле data.

Вторая иерархия является иерархией классов-*итераторов*; она включает абстрактный класс Iterator и классы ConcreteIteratorA, ConcreteIteratorB и ConcreteIteratorC. Класс Iterator содержит четыре абстрактных метода без параметров: First и Next (не возвращают значений), IsDone (возвращает логическое значение), CurrentItem (возвращает целочисленное значение).

Метод First устанавливает итератор на первый элемент перебираемого набора данных, метод Next переводит итератор на следующий элемент (или за конец набора), метод IsDone возвращает значение true, если итератор указывает на позицию за концом набора, и false, если итератор указывает на некоторый элемент набора; метод CurrentItem возвращает элемент набора, на который указывает итератор, или −1, если итератор находится за последним элементом набора (возможен также вариант, когда в последней ситуации возбуждается исключение, поскольку в программе такая ситуация обычно свидетельствует об ошибке). Для пустого набора метод First сразу устанавливает итератор за конец набора; при нахождении итератора за концом набора метод Next не выполняет никаких действий.

Конкретные классы-итераторы A, B, C связаны с ранее описанными классами-агрегатами A, B, C и обеспечивают особый способ перебора содержащихся в них данных. Итератор A перебирает все *цифры* целочисленного поля data агрегата A *в обратном порядке* (знак числа игнорируется); для числа 0 возвращается цифра 0 (это единственная ситуация, когда последним элементом набора является цифра 0). Итератор B перебирает все *цифровые символы* строкового поля data агрегата B *в обратном порядке*. Итератор C перебирает все цифры всех элементов структуры data агрегата C, причем как сами элементы, так и их цифры должны перебираться *в обратном порядке*. Для итераторов B и C возможна ситуация, когда перебираемый набор является пустым (если строковое поле data не содержит цифровых символов или структура data не содержит ни одного элемента). Каждый конкретный итератор содержит поле aggr, которое является ссылкой на связанный с ним объект-агрегат; это поле инициализируется в конструкторе итератора с помощью соответствующего параметра. Кроме того, итераторы содержат вспомогательные поля, используемые при реализации методов First, Next, IsDone и CurrentItem.

Дано целое число N (≤ 10) и N наборов элементов. Первый элемент каждого набора представляет собой символ "A", "B" или "C"; он определяет тип создаваемого объекта-агрегата (A, B или C). Следующие элементы каждого набора определяют поле data создаваемого агрегата: для агрегата A это одно целое число, для агрегата B — одна строка, для агрегата C — целое число K (≤ 10), определяющее размер структуры данных data, и K целых чисел — элементов этой структуры (число K может быть равно 0).

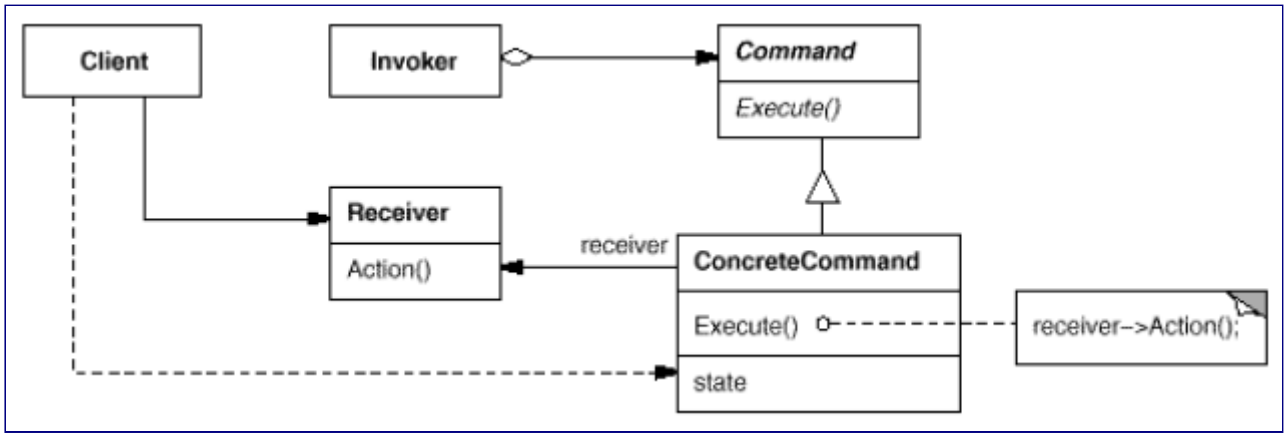
Описать структуру (например, массив) из N элементов-ссылок типа Aggregate, сохранить в ней исходные объекты-агрегаты и выполнить обработку этих объектов, перебирая их *в обратном порядке*. Для каждого объекта-агрегата требуется вывести сумму цифр, возвращаемых его итератором (или 0, если итератор ничего не возвращает), а затем — сами цифры, возвращаемые его итератором.

Примечание. В языках с C-подобным синтаксисом для перебора элементов с применением итератора it, имеющего описанный выше набор методов, можно использовать следующий вариант цикла for: `for (it.First(); !it.IsDone(); it.Next()) <обработка it.CurrentItem()>`

```
public abstract class Aggregate
{
    public abstract Iterator CreateIterator();
}

// Implement the ConcreteAggregateA, ConcreteAggregateB
//   and ConcreteAggregateC descendant classes

public abstract class Iterator
{
    public abstract void First();
```

```
        public abstract void Next();
        public abstract bool IsDone();
        public abstract int CurrentItem();
    }

    // Implement the ConcreteIteratorA, ConcreteIteratorB
    //   and ConcreteIteratorC descendant classes
```

OOP3Behav8°.



Command (**Команда**) — паттерн поведения.

Известен также под именем **Action** (**Действие**), **Transaction** (**Транзакция**).

Частота использования: выше средней.

Назначение: инкапсулирует запрос (действие, операцию) как объект, позволяя тем самым задавать параметры клиентов для обработки соответствующих запросов, ставить запросы в очередь или протоколировать их, а также поддерживать отмену операций.

Участники:

• *Command* (*Команда*) — объявляет интерфейс для выполнения запроса;

• *ConcreteCommand* (*Конкретная команда*) — определяет связь между объектом-получателем Receiver и требуемым запросом; реализует метод Execute путем вызова требуемых методов объекта Receiver;

• *Client* (*Клиент*) — создает объект класса ConcreteCommand и задает его получателя;

• *Invoker* (*Инициатор*) — обращается к команде для выполнения запроса; не использует никакой информации о конкретном получателе запроса;

• *Receiver* (*Получатель*) — располагает информацией об операциях, необходимых для выполнения запроса; в роли получателя может выступать любой класс.

Основной особенностью паттерна Command является то, что он отделяет объект-инициатор Invoker, выдающий запросы, от объекта-получателя Receiver, который умеет эти запросы выполнять.

Задание 1. Реализовать классы, связанные с организацией запросов на основе паттерна Command. Иерархия классов-*команд* включает абстрактный класс Command с абстрактным методом Execute (который не имеет параметров и ничего не возвращает) и классы ConcreteCommandA и ConcreteCommandB, связанные с конкретными командами A и B. Классы ConcreteCommandA и ConcreteCommandB включают поле recv, определяющее получателя соответствующей команды; это поле является ссылкой на объект класса ReceiverA для команды A и ссылкой на объект класса ReceiverB для команды B; поле recv определяется в конструкторе конкретной команды, имеющем параметр recv соответствующего типа. Метод Execute конкретной команды A вызывает метод ActionA для объекта recv (типа ReceiverA), метод Execute конкретной команды B вызывает метод ActionB для объекта recv (типа ReceiverB).

Классы-*получатели* ReceiverA и ReceiverB содержат поле cli — ссылку на объект класса Client и поле info строкового типа; эти поля инициализируются в конструкторе, имеющем одноименные параметры cli и info. Метод ActionA класса ReceiverA вызывает метод AddLeft(info) объекта cli; метод ActionB класса ReceiverB вызывает метод AllRight(info)

объекта cli. Следует подчеркнуть, что классы-получатели не входят в какую-либо особую иерархию, и каждый из них реализует свой собственный набор методов.

Класс-*клиент* Client содержит строковое поле info, которое инициализируется пустой строкой в конструкторе (конструктор не имеет параметров). Класс Client также содержит три метода: AddLeft(newInfo), AddRight(newInfo) и GetInfo. Методы AddLeft и AddRight имеют строковый параметр newInfo и добавляют строку newInfo соответственно в начало и конец поля info; эти методы ничего не возвращают. Метод GetInfo без параметров возвращает значение поля info.

Класс-*инициатор* Invoker предназначен для выполнения связанной с ним команды. Он содержит поле cmd — ссылку на объект типа Command, которая инициализируется в конструкторе с одноименным параметром cmd, а также метод Invoke, в котором выполняется вызов метода Execute команды cmd.

Дано целое число N (≤ 10), задающее количество различных команд, и набор S из N различных строк, каждая из которых начинается либо с символа "A", либо с символа "B". Создать объект cli типа Client и набор cmd из N команд (например, массив) с элементами-ссылками типа Command. Каждый элемент набора cmd является либо командой ConcreteCommandA (если соответствующая строка набора S начинается с символа "A"), либо командой ConcreteCommandB (если соответствующая строка набора S начинается с символа "B"); при создании команд A и B используются объекты типа ReceiverA или ReceiverB соответственно, которые, в свою очередь, создаются с помощью конструкторов, имеющих следующие параметры: ранее созданный объект cli типа Client и соответствующая строка из набора S. Например, если очередной строкой набора S является строка "Apqr", то соответствующим элементом набора cmd должен быть объект ConcreteCommandA, причем его конструктору должен передаваться объект ReceiverA, в конструкторе которого указываются параметры cli и "Apqr".

Также дано целое число K (≤ 30), задающее количество различных инициаторов (объектов типа Invoker), и набор из K целых чисел со значениями из диапазона от 0 до N − 1 (каждый элемент набора определяет *индекс* некоторой команды из набора cmd). Создать набор inv из K инициаторов (например, массив) с элементами-ссылками типа Invoker и инициализировать каждого инициатора командой с соответствующим индексом из набора cmd (например, если начальным элементом набора из K целых чисел является число 5, то начальный инициатор inv[0] должен инициализироваться командой cmd[5]). Несколько инициаторов может быть связано с одной и той же командой (что является стандартной ситуацией при организации пользовательского интерфейса, когда одну и ту же команду можно выполнить, например, с помощью пункта меню, кнопки быстрого доступа или горячей клавиши).

Наконец, дано целое число M (≤ 20), задающее количество команд для выполнения, и набор из M целых чисел со значениями из диапазона от 0 до K − 1 (каждый элемент набора определяет *индекс* того инициатора из набора inv, который должен использоваться для выполнения требуемой команды). Выполнить требуемые команды, вызвав метод Invoke для элементов набора inv с указанными индексами. После выполнения каждой команды выводить текущее состояние объекта cli, используя его метод GetInfo.

```
// Implement the Client, ReceiverA and ReceiverB classes

public abstract class Command
{
    public abstract void Execute();
}

// Implement the ConcreteCommandA
//   and ConcreteCommandB descendant classes

public class Invoker
{
    Command cmd;
    public Invoker(Command cmd)
    {
        this.cmd = cmd;
    }
```

```
    public void Invoke()
    {
        cmd.Execute();
    }
}
```

OOP3Behav9°. **Command** (**Команда**) — паттерн поведения.

Задание 2. Реализовать набор классов для варианта паттерна Command с дополнительными возможностями, связанными с *созданием макрокоманд* и *отменой предыдущих действий*.

Имеются три класса-*получателя*, реализующих различные операции и умеющих отменять их: класс OperationA включает статические методы ActionA (выводит строку "+A") и UndoActionA (выводит строку "−A"), класс OperationB включает статические методы ActionB (выводит строку "+B") и UndoActionB (выводит строку "−B"), класс OperationC включает статические методы ActionC (выводит строку "+C") и UndoActionC (выводит строку "−C"). Других методов или полей классы-получатели не содержат. Следует подчеркнуть, что эти классы не входят в какую-либо особую иерархию.

Иерархия классов-*команд* начинается с абстрактного класса Command, включающего методы Execute и Unexecute (методы не имеют параметров и не возвращают значений). Его потомками являются классы CommandA, CommandB, CommandC и MacroCommand. Конструкторы классов CommandA, CommandB, CommandC не имеют параметров и не выполняют дополнительных действий. Метод Execute команды A выполняет вызов статического метода ActionA класса-получателя OperationA, метод Unexecute команды A выполняет вызов статического метода UndoActionA. Методы Execute и Unexecute команд B и C определяются аналогично, с использованием статических методов классов OperationB и OperationC.

Класс MacroCommand позволяет объединять имеющиеся команды в последовательности команд (*макрокоманды*). Он содержит структуру cmds (например, массив) с элементами-ссылками типа Command, которая инициализируется в конструкторе, имеющем соответствующий параметр-структуру (можно считать, что макрокоманда содержит не более 5 команд). Метод Execute класса MacroCommand выполняет вызов методов Execute всех элементов структуры cmds в исходном порядке, а метод Unexecute — вызов методов Unexecute всех элементов структуры cmds в обратном порядке.

Реализовать класс Menu, предоставляющий средства для настройки *инициаторов* команд, их выполнения, а также выполнения операций отмены и восстановления. Класс Menu содержит две структуры: массив availCmds размера 3, в котором хранятся ссылки на команды, *доступные для выполнения*, и структуру lastCmds, в котором хранятся ссылки на *ранее выполненные команды*, что дает возможность *отменять* эти команды или, после отмены, *восстанавливать* их. В качестве структуры lastCmds удобно использовать динамическую структуру, позволяющую добавлять в конец новые элементы и удалять часть последних элементов. Можно считать, что структура lastCmds в любой момент времени будет содержать не более 40 элементов. Со структурой lastCmds связано дополнительное целочисленное поле undoIndex, определяющее индекс элемента из lastCmd, после которого следуют *ранее отмененные* команды (которые впоследствии могут быть восстановлены). Конструктор класса Menu содержит два ссылочных параметра: cmd1 и cmd2 типа Command; эти параметры определяют два начальных элемента массива availCmds; третий элемент этого массива является макрокомандой (объектом типа MacroCommand), включающей команды cmd1 и cmd2 в указанном порядке.

Класс Menu содержит три метода: Invoke(cmdIndex), Undo(count) и Redo(count); методы имеют целочисленные параметры и не возвращают значений.

Метод Invoke(cmdIndex) *выполняет* команду из массива availCmds с индексом cmdIndex (при реализации этого метода можно считать, что параметр cmdIndex всегда находится в допустимом диапазоне 0–2). Кроме того, при выполнении метода Invoke из структуры lastCmds удаляются все конечные элементы, начиная с элемента с индексом undoIndex + 1 (если такие элементы существуют), в конец структуры lastCmds добавляется ссылка на только что выполненную команду, а значение поля undoIndex полагается равным индексу добавленной команды.

Метод Undo(count) *отменяет* count выполненных команд, хранящихся в структуре lastCmds, начиная с команды с индексом undoIndex в направлении *уменьшения* индексов (если в структуре lastCmds содержится недостаточно элементов, то отменяются все доступные команды). Для каждой из этих команд вызывается метод Unexecute; кроме того, значение поля undoIndex корректируется так, чтобы оно соответствовало последней еще не отмененной команде (если отменены все команды из набора lastCmds, то значение undoIndex полагается равным −1).

Метод Redo(count) *восстанавливает* count ранее отмененных команд, выполняя метод Execute для элементов структуры lastCmds, начиная с команды с индексом undoIndex + 1 в направлении *увеличения* индексов (если в структуре lastCmds содержится недостаточно элементов, то восстанавливаются все ранее отмененные команды). Кроме того, значение поля undoIndex корректируется так, чтобы оно соответствовало последней восстановленной команде.

Описанный механизм отмены/восстановления команд позволяет отменять и восстанавливать любое количество ранее выполненных команд (с сохранением их исходного порядка выполнения), однако при выполнении новой команды он блокирует возможность восстановления ранее отмененных команд (поскольку их результат может конфликтовать с результатом выполнения новой команды).

Даны два различных символа C1 и C2, которые могут принимать три значения: "A", "B", "C". Символ C1 определяет тип первого параметра конструктора объекта Menu (значение "A" соответствует классу CommandA, значение "B" — классу CommandB, значение "C" — классу CommandC). Аналогичным образом, символ C2 определяет тип второго параметра конструктора объекта Menu. Используя указанную информацию, создать объект m типа Menu.

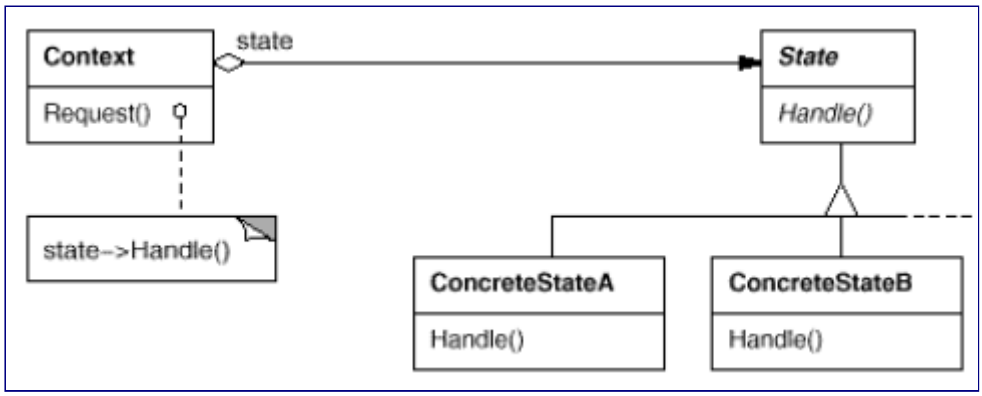
Также дано целое число N (≤ 40), задающее количество методов объекта m для выполнения, и набор из N двухсимвольных строк, кодирующих требуемые методы. Первый символ каждой строки является одной из букв "I", "U", "R", а второй символ является цифрой, причем в случае первого символа "I" возможны только цифры "0", "1", "2", а в случае символов "U" и "R" — только цифры в диапазоне от "1" до "9". Строки "I0", "I1", "I2" соответствуют методу Invoke с параметрами 0, 1, 2; строки, начинающиеся с символа "U", соответствуют методу Undo, причем цифра определяет количество команд для отмены (например, строка "U4" соответствует методу Undo(4)); строки, начинающиеся с символа "R", соответствуют методу Redo, а цифра определяет количество восстанавливаемых команд.

Выполнить в указанном порядке все методы для созданного объекта m типа Menu. Выводить какие-либо данные не требуется, так как вывод осуществляется в классах-получателях OperationA, OperationB и OperationC при выполнении соответствующих команд.

```
public static class OperationA
{
    public static void ActionA()
    {
        Put("+A");
    }
    public static void UndoActionA()
    {
        Put("-A");
    }
}

public static class OperationB
{
    public static void ActionB()
    {
        Put("+B");
    }
    public static void UndoActionB()
    {
        Put("-B");
    }
```

```
}

public static class OperationC
{
    public static void ActionC()
    {
        Put("+C");
    }
    public static void UndoActionC()
    {
        Put("-C");
    }
}

public abstract class Command
{
    public abstract void Execute();
    public abstract void Unexecute();
}

// Implement the CommandA, CommandB, CommandC
//   and MacroCommand descendant classes

public class Menu
{
    // Add required fields
    public Menu(Command cmd1, Command cmd2)
    {
        // Implement the constructor
    }
    public void Invoke(int cmdIndex)
    {
        // Implement the method
    }
    public void Undo(int count)
    {
        // Implement the method
    }
    public void Redo(int count)
    {
        // Implement the method
    }
}
```

OOP3Behav10°.



State (**Состояние**) — паттерн поведения.

Частота использования: средняя.

Назначение: позволяет объекту варьировать свое поведение в зависимости от внутреннего состояния, которое определяется одним из нескольких объектов, связанных с конкретными состояниями и имеющими одинаковый интерфейс. Извне создается впечатление, что изменился класс объекта.

Участники:

• *Context* (*Контекст*) — определяет интерфейс, представляющий интерес для клиентов; хранит экземпляр подкласса ConcreteState, которым определяется текущее состояние;

• *State* (*Состояние*) — определяет интерфейс для инкапсуляции поведения, ассоциированного с конкретным состоянием контекста Context;

• *ConcreteStateA*, *ConcreteStateB* (*Конкретные состояния*) — реализуют поведение, ассоциированное с некоторым состоянием контекста Context.

Задание 1. Реализовать набор классов, связанных с разбором текста на основе паттерна State. Предполагается, что обрабатываемый текст включает обычное содержимое (токен Normal), строковые литералы, заключенные в двойные кавычки (токен String), и комментарии, заключенные в фигурные скобки (токен Comm). Фигурные скобки в строковых литералах считаются обычными символами, как и двойные кавычки в комментариях. Если в строковом литерале встречаются две двойных кавычки подряд, то они рассматриваются как обычный символ "двойная кавычка", входящий в строковый литерал. Комментарии не являются вложенными; открывающая фигурная скобка внутри комментария рассматривается как обычный символ.

Признаком конца разбираемого текста является наличие точки, которая не считается входящей в сам текст. В токены String не включаются обрамляющие кавычки, в токены Comm не включаются обрамляющие фигурные скобки. Если последний строковый литерал или комментарий не заканчивается требуемым символом (кавычкой или фигурной скобкой соответственно), то подобный фрагмент текста считается *ошибочным токеном* ErrString или ErrComm соответственно; такой токен должен содержать символы от начала строкового литерала или комментария вплоть до завершающей точки (не включая эту точку).

Любые виды токенов могут быть пустыми; в начале и конце текста, а также между специальными токенами String и Comm обязательно присутствует токен Normal (возможно, пустой). Исключением являются ошибочные токены ErrString и ErrComm, после которых разбор текста завершается.

Иерархия классов-*состояний* включает абстрактный класс State с абстрактным методом GetNextToken (не имеет параметров, возвращает строку с описанием очередного токена разбираемого текста) и классы ConcreteStateNormal, ConcreteStateString, ConcreteStateComm и ConcreteStateFinal. Каждый конкретный класс, кроме класса ConcreteStateFinal, содержит поле ct — ссылку на объект Context — и поле index целого типа, которые инициализируются в конструкторе с использованием соответствующих параметров. Поле ct определяет объект, содержащий разбираемый текст, а поле index определяет индекс позиции, начиная с которой требуется продолжить разбор текста.

Метод GetNextToken возвращает строку, содержащую полученный токен (возможно, пустой), перед которым указывается его тип и двоеточие (например, "Normal:abc", "ErrString:mn2", "Comm:"). Класс ConcreteStateFinal не содержит полей, его конструктор не выполняет дополнительных действий, а метод GetNextToken всегда возвращает пустую строку. После определения текущего токена Normal метод GetNextToken класса ConcreteStateNormal вызывает метод SetState контекста ct, указывая в качестве параметра экземпляр класса ConcreteStateString (если обнаружен символ "двойная кавычка"), ConcreteStateComm (если обнаружен символ "{") или ConcreteStateFinal (если обнаружен символ "точка"). После определения текущего *правильного* токена String или Comm метод GetNextToken классов ConcreteStateString и ConcreteStateComm вызывает метод SetState контекста ct, указывая в качестве параметра экземпляр класса ConcreteStateNormal. После определения *ошибочного* токена ErrString или ErrComm метод GetNextToken классов ConcreteStateString и ConcreteStateComm вызывает метод SetState контекста ct с экземпляром класса ConcreteStateFinal.

Класс-*контекст* Context содержит строковое поле text с разбираемым текстом и поле-ссылку currentState типа State. Конструктор класса имеет параметр text, используемый для инициализации поля text; поле currentState инициализируется объектом типа ConcreteStateNormal. Класс Context имеет методы GetCharAt(index), SetState(newState) и GetNextToken. Метод GetCharAt возвращает символ поля text с индексом index (предполагается, что индекс находится в допустимом диапазоне); метод SetState изменяет поле currentState, присваивая ему значение параметра newState (этот метод, наряду с

методом GetCharAt, используется в методах классов-состояний); метод GetNextToken возвращает очередной токен разбираемого текста, вызывая одноименный метод объекта currentState.

Дана строка, которая оканчивается точкой. Используя объект ct типа Context, выполнить разбор данной строки, вызывая метод GetNextToken объекта ct и выводя его возвращаемый результат, пока очередной вызов не вернет пустую строку (пустую строку выводить не следует).

```
public abstract class State
{
    public abstract string GetNextToken();
}

// Implement the ConcreteStateNormal, ConcreteStateString,
//   ConcreteStateComm and ConcreteStateFin descendant classes

// Implement the Context class
```

OOP3Behav11°. **State** (**Состояние**) — паттерн поведения.

Задание 2. Реализовать набор классов, связанных с моделированием работы автомата по продаже шариков (ball machine).

Иерархия классов-состояний включает абстрактный класс State с абстрактными методами, описывающими возможные действия с автоматом — InsertCoin (вложить монетку), GetBall (получить шарик), ReturnCoin (вернуть монетку), AddBall (добавить шарик в автомат), и набор конкретных классов, соответствующих различным возможным состояниям автомата: ReadyState (автомат содержит шарики и готов к приему монетки), HasPayedState (автомат получил монетку и готов выдать шарик или вернуть монетку), NoBallState (в автомате нет шариков). Все методы не имеют параметров и не возвращают значения.

Каждый конкретный класс содержит поле machine — ссылку на объект типа BallMachine; это поле инициализируется в конструкторе, имеющем одноименный параметр. Все переопределенные методы в каждом конкретном классе могут выводить соответствующее текстовое сообщение, изменять состояние автомата, вызывая для объекта machine его метод SetState с подходящим параметром, и выполнять другие действия. Ниже перечисляются действия всех методов для конкретных классов-состояний.

Класс ReadyState:

• метод InsertCoin выводит текст "Coin is inserted" (монетка получена) и переводит автомат в состояние HasPayedState;

• методы GetBall и ReturnCoin выводят текст "You need to pay first" (вначале заплатите);

• метод AddBall не выполняет никаких действий.

Класс HasPayedState:

• метод InsertCoin выводит текст "You have already paid" (вы уже заплатили);

• метод ReturnCoin выводит текст "Take your coin" (получите вашу монетку) и переводит автомат в состояние ReadyState;

• метод AddBall, как и для предыдущего класса, не выполняет никаких действий;

• метод GetBall выводит текст "Take your ball" (получите ваш шарик) и, кроме того, вызывает метод DecreaseBallCount объекта machine и анализирует его возвращаемое значение (равное оставшемуся количеству шариков): если это значение больше нуля, то автомат переводится в состояние ReadyState, в противном случае автомат переводится в состояние NoBallState.

Класс NoBallState:

• методы InsertCoin, GetBall и ReturnCoin выводят текст "Sorry, balls are over" (извините, шарики закончились);

• метод AddBall переводит автомат в состояние ReadyState и не выводит сообщений.

Класс BallMachine содержит поле ballCount целого типа (равное текущему количеству шариков), поля ready, hasPayed и noBall типа State (в которых содержатся ссылки на соответствующие объекты-состояния), а также поле currentState типа State (в котором

содержится ссылка на текущее состояние). Поля инициализируются в конструкторе без параметров; полю ballCount присваивается значение 3, а для инициализации полей ready, hasPayed и noBall используются конструкторы соответствующих классов с параметром — ссылкой на создаваемый объект типа BallMachine. Кроме того, в конструкторе выполняется присваивание полю currentState значения ready.

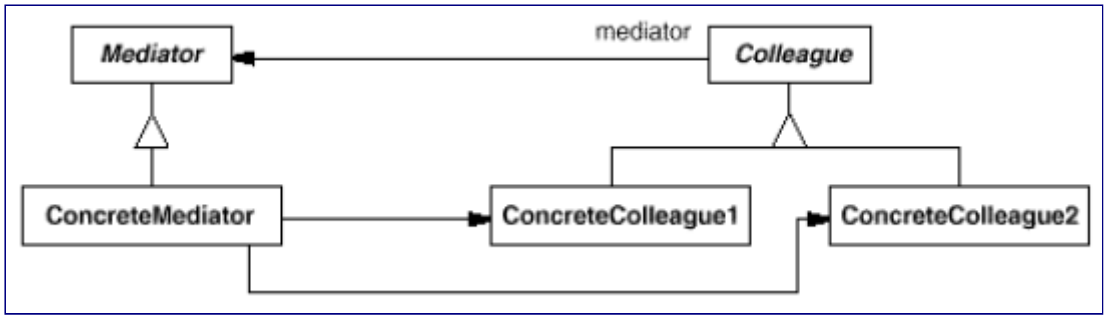
Класс BallMachine содержит методы InsertCoin, GetBall, ReturnCoin, AddBall, в которых выполняется вызов одноименных методов объекта currentState; в методе AddBall дополнительно выводится текст "Ball is added" (шарик добавлен) и выполняется увеличение на 1 поля ballCount. В классе также надо реализовать метод DecreaseBallCount без параметров, который уменьшает на 1 поле ballCount и возвращает новое значение этого поля (метод DecreaseBallCount используется в методе GetBall класса HasPayedState), и метод SetState(newState), который присваивает полю currentState значение параметра newState. Для доступа на чтение к полям ready, hasPayed и noBall надо предусмотреть методы GetReadyState, GetHasPayedState и GetNoBallState (эти методы, совместно с методом SetState, используются в методах классов-состояний для изменения состояния объекта BallMachine).

Дана строка S, содержащая только символы "I", "G", "R", "A"; каждый символ соответствует одной из команд автомата BallMachine: "I" — InsertCoin, "G" — GetBall, "R" — ReturnCoin, "A" — AddBall. Создать объект типа BallMachine и вызвать для него набор команд, соответствующий символам исходной строки S в порядке их следования в строке. Выводить какие-либо результаты не требуется, так как вывод осуществляется в методах, реализующих команды автомата.

```
public abstract class State
{
    public abstract void InsertCoin();
    public abstract void GetBall();
    public abstract void ReturnCoin();
    public abstract void AddBall();
}

// Implement the ReadyState, HasPayedState
//   and NoBallState descendant classes

// Implement the BallMachine class
```

Mediator, Chain of Responsibility, Visitor, Interpreter

OOP3Behav12°.



Mediator (**Посредник**) — паттерн поведения.

Частота использования: ниже средней.

Назначение: определяет объект, инкапсулирующий способ взаимодействия множества объектов. Посредник обеспечивает слабую связанность системы, избавляя объекты от необходимости явно ссылаться друг на друга и позволяя тем самым независимо изменять способы взаимодействия между ними.

Участники:

• *Mediator* (*Посредник*) — определяет интерфейс для обмена информацией с объектами Colleague;

• *ConcreteMediator* (*Конкретный посредник*) — реализует кооперативное поведение, координируя действия объектов Colleague; владеет информацией о коллегах;

• *Colleague* (*Коллега*) — определяет интерфейс для взаимодействия с посредником;

• *ConcreteColleagueA*, *ConcreteColleagueB* (*Конкретные коллеги*) — знают о своем объекте Mediator; обмениваются информацией только с посредником (вместо того, чтобы общаться между собой напрямую).

Посредник реализует кооперативное поведение путем переадресации каждого запроса, посланного ему каким-либо коллегой, подходящему коллеге (одному или нескольким).

Задание 1. Реализовать две иерархии классов, связанные с применением паттерна Mediator. Первая иерархия является иерархией классов-*коллег* и включает абстрактный класс Colleague и конкретные классы ConcreteColleague1 и ConcreteColleague2.

Класс Colleague содержит поле m, являющееся ссылкой на связанный с данным коллегой объект Mediator, и методы SetMediator(m) (не возвращает значения, инициализирует поле m одноименным параметром метода, также имеющим тип Mediator) и Notify (не имеет параметров и не возвращает значения, извещает посредник m о наступлении события путем вызова метода NotifyFrom объекта m, причем в качестве параметра метода NotifyFrom передается ссылка на объект, вызвавший метод Notify). В методе Notify перед вызовом метода NotifyFrom можно проверять, что поле m не является пустой ссылкой, хотя при согласованной работе коллег и посредников такая ситуация не должна возникать. Методы SetMediator и Notify не являются абстрактными, они реализуются в классе Colleague и наследуются всеми подклассами без каких-либо изменений.

Класс ConcreteColleague1 дополнительно содержит целочисленное поле data (инициализируется в конструкторе значением 1), класс ConcreteColleague2 содержит строковое поле data (инициализируется в конструкторе строкой "ab"). Конкретные классы также содержат методы GetData и SetData, позволяющие обращаться к полю data на чтение и запись соответственно. Конструкторы классов ConcreteColleague1 и ConcreteColleague2 не имеют параметров.

Вторая иерархия является иерархией классов-*посредников*, связанных с ранее описанными классами-коллегами. Она включает абстрактный класс Mediator и конкретные классы ConcreteMediatorA и ConcreteMediatorB. Класс Mediator содержит абстрактный метод NotifyFrom(coll), имеющий параметр-ссылку типа Colleague и не возвращающий значений. Этот метод информирует посредник о событии, наступившем в объекте coll. Напомним, что именно этот метод должен вызываться в методе Notify каждого конкретного класса-коллеги.

Классы ConcreteMediatorA и ConcreteMediatorB реализуют два различных сценария взаимодействия объектов типа ConcreteColleague1 и ConcreteColleague2. Они создают набор взаимодействующих коллег, позволяют получать ссылки на них и обрабатывают связанные с ними события, изменяя состояние других коллег. Конструкторы классов ConcreteMediatorA и ConcreteMediatorB не имеют параметров.

Класс ConcreteMediatorA содержит поля c1 (типа ConcreteColleague1) и c2 (типа ConcreteColleague2). Объекты c1 и c2 создаются в конструкторе класса ConcreteMediatorA и связываются с объектом-посредником путем вызова методов SetMediator (с параметром, являющимся ссылкой на создаваемый объект ConcreteMediatorA). Для доступа к объектам c1 и c2 предусмотрены методы GetC1 и GetC2, возвращающие ссылки на соответствующие объекты, приведенные к типу Colleague. Метод NotifyFrom(coll) класса ConcreteMediatorA выполняет следующие действия:

• если параметр coll является ссылкой на поле c1, то к строковому полю data объекта c2 добавляется пробел и строковое представление поля data объекта c1;

• если параметр coll является ссылкой на поле c2, то числовое поле data объекта c1 увеличивается на текущую длину строкового поля data объекта c2.

В конце своей работы метод NotifyFrom выводит на экран текущие значения полей data объектов c1 и c2.

Класс ConcreteMediatorB содержит поля c1a и c1b (типа ConcreteColleague1) и c2 (типа ConcreteColleague2). Объекты c1a, c1b и c2 создаются в конструкторе класса ConcreteMediatorB (действия при их создании аналогичны действиям, описанным для класса ConcreteMediatorA). Для доступа к объектам предусмотрены методы GetC1a, GetC1b и GetC2, возвращающие ссылки на соответствующие объекты, приведенные к типу

Colleague. Метод NotifyFrom(coll) класса ConcreteMediatorB выполняет следующие действия:

• если параметр coll является ссылкой на поле c1a, то числовое поле data объекта c1b увеличивается на значение поля data объекта c1a, а к строковому полю data объекта c2 добавляется символ "a";

• если параметр coll является ссылкой на поле c1b, то числовое поле data объекта c1a увеличивается на значение поля data объекта c1b, а к строковому полю data объекта c2 добавляется символ "b";

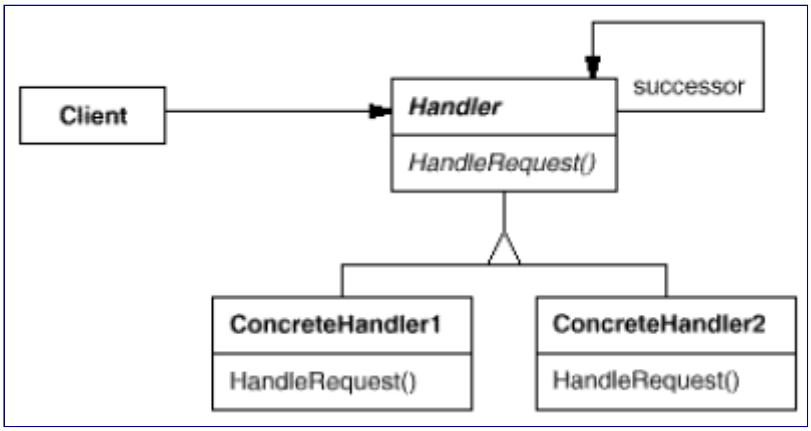
• если параметр coll является ссылкой на поле c2, то в поля data объектов c1a и c1b записываются числа, равные количеству символов "a" и "b" в строке data объекта c2 (например, если строка равна "abaab", то в поле data объекта c1a записывается число 3, а в поле data объекта c1b записывается число 2).

В конце своей работы метод NotifyFrom выводит на экран текущие значения полей data объектов c1a, c1b и c2.

Создать объекты-посредники ma (типа ConcreteMediatorA) и mb (типа ConcreteMediatorB). Также создать структуру coll (например, массив) из пяти ссылок на элементы типа Colleague и записать в нее ссылки, возвращаемые следующими методами (в указанном порядке): ma.GetC1, ma.GetC2, mb.GetC1a, mb.GetC1b, mb.GetC2. Дано целое число N (≤ 20) и набор из N целых чисел в диапазоне от 0 до 4, определяющих индексы элементов структуры coll. Для каждого числа K из данного набора выполнить вызов метода Notify для элемента структуры coll с индексом K. Выводить какие-либо результаты не требуется, так как вывод осуществляется в методах NotifyFrom объектов-посредников.

```
public abstract class Colleague
{
    Mediator m;
    public void SetMediator(Mediator m)
    {
        this.m = m;
    }
    public void Notify()
    {
        m.NotifyFrom(this);
    }
}

// Implement the ConcreteColleague1
//   and ConcreteColleague2 descendant classes

public abstract class Mediator
{
    public abstract void NotifyFrom(Colleague coll);
}

// Implement the ConcreteMediatorA
//   and ConcreteMediatorB descendant classes
```

OOP3Behav13°.



Chain of Responsibility (Цепочка обязанностей) — паттерн поведения.

Частота использования: ниже средней.

Назначение: позволяет избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким объектам. Связывает объекты-получатели в *цепочку* и передает запрос вдоль этой цепочки, пока его не обработают.

Участники:

• *Handler* (*Обработчик*) — определяет интерфейс для обработки запросов; может (но не обязан) реализовывать связь с преемником;

• *ConcreteHandler* (*Конкретный обработчик*) — обрабатывает запрос, за который отвечает; имеет доступ к своему преемнику, которому направляет запрос, если не может его обработать самостоятельно;

• *Client* (*Клиент*) — отправляет запрос некоторому объекту ConcreteHandler в цепочке.

Задание 1. Реализовать иерархию классов-*обработчиков*, включающую абстрактный класс Handler и два конкретных класса ConcreteHandler1 и ConcreteHandler2. Класс Handler содержит абстрактный метод HandleRequest(req) (не возвращает значений, имеет параметр req целого типа, определяющий номер запроса).

Класс ConcreteHandler1 имеет поле successor — ссылку на объект Handler и три целочисленных поля: id (целочисленный идентификатор обработчика), req1 и req2 (определяют диапазон номеров запросов, которые может обработать данный обработчик). Все эти поля инициализируются в конструкторе с использованием параметров конструктора successor, id, req1, req2. Метод HandleRequest(req) класса ConcreteHandler1 выполняет следующие действия. Если его параметр req лежит в диапазоне от req1 до req2, то запрос обрабатывается путем вывода текста "Request <req> processed by handler <id>" (запрос <req> обработан обработчиком <id>), где на позициях <req> и <id> указываются значения параметра req и поля id. В противном случае выполняется вызов метода HandleRequest(req) для объекта successor (перед этим вызовом можно проверять, что поле successor не является пустой ссылкой, хотя при правильном построении цепочки обработчиков такая ситуация не должна возникать).

Класс ConcreteHandler2 является особым *терминальным* обработчиком, который должен находиться в конце цепочки обработчиков и к которому, таким образом, поступают все необработанные запросы. Он не имеет полей, его конструктор не выполняет дополнительных действий, а метод HandleRequest(req) выводит на экран текст "Request <req> not processed" (запрос <req> не обработан), где на позиции <req> указывается значение параметра req.

Вспомогательный класс Client содержит поле h — ссылку типа Handler на обработчик, являющийся первым в ранее сформированной цепочке обработчиков. Поле h инициализируется в конструкторе с помощью одноименного параметра. Класс Client также содержит метод SendRequest(req), имеющий целочисленный параметр req и не возвращающий значения; в этом методе выполняется вызов метода HandleRequest(req) для объекта h.

Дано целое число N (≤ 10) и N пар целых чисел (r1, r2) (для любой пары выполняется неравенство r1 ≤ r2). Используя переменную h — ссылку на объект Handler, последовательно создать один объект типа ConcreteHandler2 и N объектов типа ConcreteHandler1. Ссылки на создаваемые объекты сохраняются в одной и той же переменной h; параметрами конструктора для объектов ConcreteHandler1 должны быть значения h, i, r1, r2, где i — индекс пары из исходного набора (пары индексируются от 0), а r1 и r2 — первый и второй элемент этой пары. В результате будет создана цепочка из N + 1 обработчика, причем первым элементом этой цепочки (ссылка на который будет храниться в переменной h) будет обработчик типа ConcreteHandler1 с идентификатором N − 1, предпоследним — обработчик типа ConcreteHandler1 с идентификатором 0, а последним — обработчик типа ConcreteHandler2. Создать объект cli типа Client, указав в качестве параметра его конструктора значение ссылки h.

Также дано целое число K (≤ 20) и набор из K различных целых чисел — номеров запросов. Для каждого запроса req из данного набора выполнить вызов метода SendRequest(req) объекта cli. Выводить какие-либо результаты не требуется, так как вывод осуществляется в методах HandleRequest объектов-обработчиков.

```
public abstract class Handler
{
    public abstract void HandleRequest(int req);
}

// Implement the ConcreteHandler1
//   and ConcreteHandler2 descendant classes

public class Client
{
    Handler h;
    public Client(Handler h)
    {
        this.h = h;
    }
    public void SendRequest(int req)
    {
        h.HandleRequest(req);
    }
}
```

OOP3Behav14°. **Chain of Responsibility** (**Цепочка обязанностей**) — паттерн поведения.

В данном задании рассматривается вариант иерархии классов обработчиков, в которой базовый класс обеспечивает в методе HandleRequest всю необходимую функциональность для передачи запроса по цепочке обработчиков, а подклассы расширяют эту функциональность. Кроме того, в задании рассматривается вариант представления запросов в виде иерархии классов, инкапсулирующих параметры запроса.

Задание 2. Реализовать две иерархии классов, связанные с применением паттерна Chain of Responsibility. Первая иерархия является иерархией классов-*запросов* и включает абстрактный класс Request и два конкретных класса RequestA и RequestB. Класс Request содержит абстрактный метод ToStr без параметров, возвращающий строковое описание запроса. В классах RequestA и RequestB определено поле param, задающее параметр запроса, причем для класса A параметр является целочисленным, а для класса B — строковым. Поле param инициализируется в конструкторе с помощью одноименного параметра конструктора. Также в этих классах определен метод GetParam, возвращающий значение поля param, и метод ToStr, возвращающий описание запроса, включающее тип запроса (букву "A" или "B"), двоеточие и параметр запроса (целое число для запроса A и строку для запроса B), например, "A:34", "B:sm".

Вторая иерархия является иерархией классов-*обработчиков* и включает конкретный базовый класс Handler и классы-потомки HandlerA и HandlerB. Класс Handler содержит поле successor (ссылку на объект Handler) и метод HandleRequest(req) (не возвращает значений, имеет параметр-ссылку req типа Request, определяющий вид запроса). Метод HandleRequest работает следующим образом: если поле successor не является пустой ссылкой, то вызывается метод HandleRequest(req) для объекта successor, в противном случае выводится текст "Request <req> not processed" (запрос <req> не обработан), где на позиции <req> указывается значение, возвращаемое методом ToStr параметра req. Класс Handler имеет конструктор с параметром-ссылкой successor типа Handler, который инициализирует одноименное поле. Таким образом, данный класс обеспечивает всю необходимую функциональность для организации цепочки обработчиков.

Классы HandlerA и HandlerB предназначены для обработки запросов соответствующего типа (A или B) и имеют целочисленное поле id (идентификатор обработчика) и поля param1 и param2 (определяют диапазон параметров запросов, которые может обработать данный обработчик). Тип полей param1 и param2 соответствует типу параметра обрабатываемого запроса: для класса HandlerA это целый тип, для класса HandlerB — строковый. Конструктор классов HandlerA и HandlerB имеет четыре параметра: successor типа ссылки на Handler, id целого типа, param1 и param2 типа, соответствующего типу одноименных полей. В конструкторе вызывается конструктор базового класса с параметром successor и инициализируются поля id, param1 и param2.

Метод HandleRequest(req) классов HandlerA и HandlerB выполняет следующие действия. Вначале проверяется *тип времени выполнения* параметра req, и в случае, если этот тип

соответствует типу обрабатываемого запроса (RequestA для HandlerA, RequestB для HandlerB), проверяется, лежит ли параметр param запроса req в диапазоне от param1 до param2 (строковые параметры для запроса RequestB сравниваются лексикографически). Если обе проверки являются успешными, то запрос обрабатывается путем вывода текста "Request <req> processed by handler <id>" (запрос <req> обработан обработчиком <id>), где на позиции <req> указывается значение, возвращаемое методом ToStr объекта req, а на позиции <id> указывается значение поля id обработчика. В противном случае выполняется вызов метода HandleRequest(req) базового класса (в котором либо происходит переход к следующему обработчику в цепочке, либо, при его отсутствии, выводится сообщение о невозможности обработать запрос).

Вспомогательный класс Client содержит поле h — ссылку типа Handler на обработчик, являющийся первым в ранее сформированной цепочке обработчиков. Поле h инициализируется в конструкторе с помощью одноименного параметра. Класс Client также содержит метод SendRequest(req), имеющий параметр-ссылку типа Request и не возвращающий значения; в этом методе выполняется вызов метода HandleRequest(req) для объекта h.

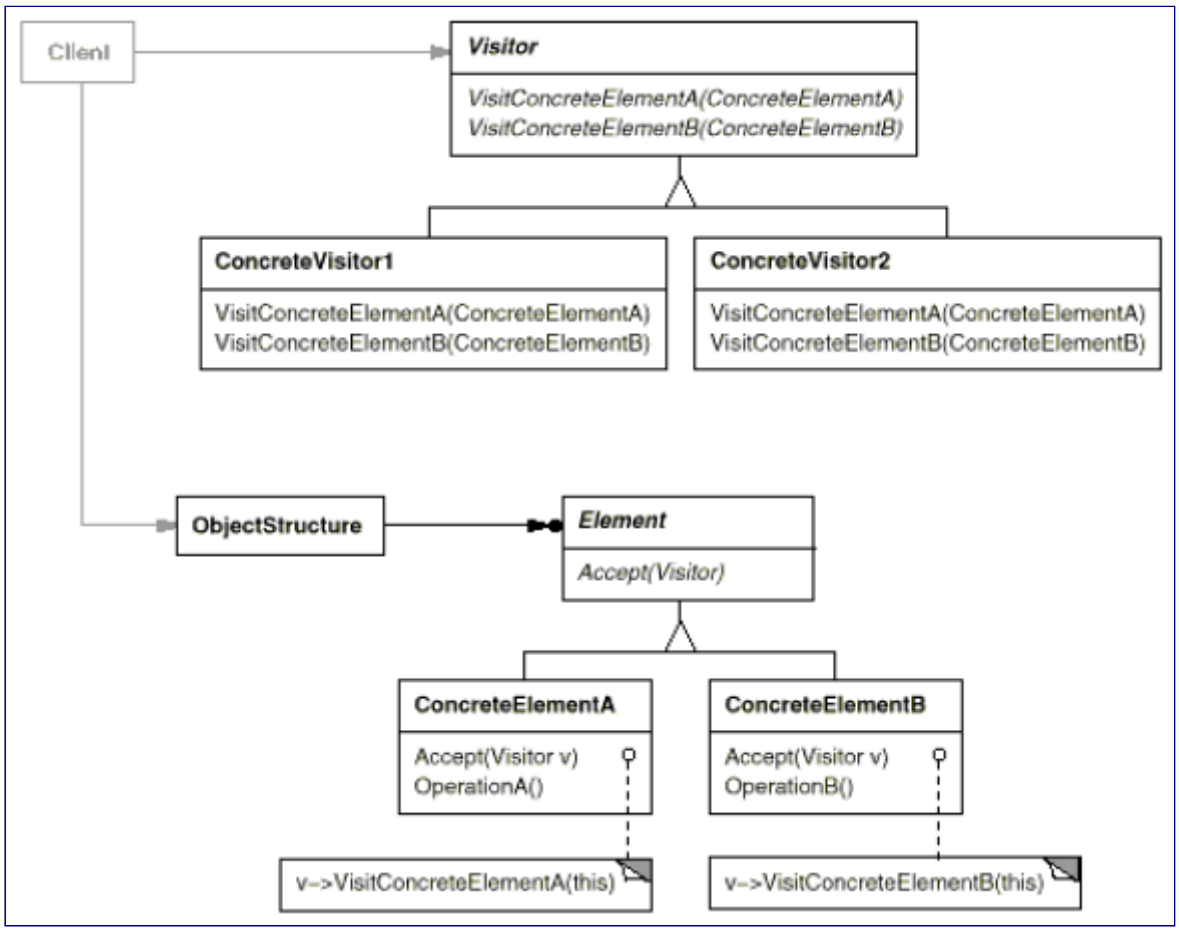
Дано целое число N (≤ 10) и N троек элементов. В каждой тройке первый элемент является символом "A" или "B", а тип остальных двух элементов (p1, p2) зависит от символа: в случае символа "A" это целые числа, а в случае "B" — строки. В любом случае выполняется неравенство p1 ≤ p2, где для чисел используется обычное сравнение, а для строк — лексикографическое. Используя переменную h — ссылку на объект Handler, последовательно создать один объект типа Handler (передав ему в качестве параметра пустую ссылку) и N объектов типа HandlerA или HandlerB (тип определяется первым элементом соответствующей тройки). Ссылки на создаваемые объекты записываются в одну и ту же переменную h; параметрами конструктора для объектов HandlerA и HandlerB должны быть значения h, i, p1, p2, где i — индекс очередной тройки из исходного набора (тройки индексируются от 0), а p1 и p2 — второй и третий элемент этой тройки. В результате будет создана цепочка из N + 1 обработчика, причем первым элементом этой цепочки (ссылка на который будет храниться в переменной h) будет обработчик типа HandlerA или HandlerB с идентификатором N − 1, предпоследним — обработчик типа HandlerA или HandlerB с идентификатором 0, а последним — обработчик типа Handler (не имеющий идентификатора). Создать объект cli типа Client, указав в качестве параметра его конструктора значение ссылки h.

Также дано целое число K (≤ 20) и набор из K различных запросов, определяемых парами элементов (c, p), где c — символ "A" или "B", а p — параметр запроса (целочисленный в случае "A", строковый в случае "B"). Для каждой пары создать запрос req соответствующего типа и выполнить вызов метода SendRequest(req) объекта cli. Выводить какие-либо результаты не требуется, так как вывод осуществляется в методах HandleRequest объектов-обработчиков.

```
public abstract class Request
{
    public abstract string ToStr();
}

// Implement the RequestA and RequestB descendant classes

public class Handler
{
    Handler successor;
    public Handler(Handler successor)
    {
        this.successor = successor;
    }
    public virtual void HandleRequest(Request req)
    {
        // Implement the method
    }
}

// Implement the HandlerA and HandlerB descendant classes
```

```
public class Client
{
    Handler h;
    public Client(Handler h)
    {
        this.h = h;
    }
    public void SendRequest(Request req)
    {
        h.HandleRequest(req);
    }
}
```

OOP3Behav15°.



Visitor (**Посетитель**) — паттерн поведения.

Частота использования: низкая.

Назначение: описывает операцию, выполняемую с каждым объектом из некоторой структуры. Паттерн Visitor позволяет определить новую операцию, не изменяя классы этих объектов и используя различные варианты операции для объектов различных типов, входящих в одну структуру.

Участники:

• *Visitor* (*Посетитель*) — объявляет группу методов Visit, в которой для каждого класса ConcreteElement в структуре объектов предусмотрен свой метод; имя метода (например, VisitConcreteElementA) и его параметр идентифицируют объект, который вызывает данный метод для отправки посетителю соответствующего запроса (это позволяет посетителю определить, элемент какого конкретного класса он посещает, и обращаться к элементу напрямую через его интерфейс);

• *ConcreteVisitor1*, *ConcreteVisitor2* (*Конкретные посетители*) — реализуют все операции, объявленные в классе Visitor и связанные с обработкой объектов различных типов, содержащихся в обрабатываемой структуре;

• *Element* (*Элемент*) — определяет метод Accept, который принимает посетителя в качестве аргумента;

• *ConcreteElementA*, *ConcreteElementB* (*Конкретные элементы*) — реализуют метод Accept, принимающий посетителя как аргумент (как правило, в этом методе происходит вызов того метода из группы методов Visit указанного посетителя, который соответствует данному конкретному элементу);

• *ObjectStructure* (*Структура объектов*) — может перечислять свои элементы, а также предоставлять посетителю высокоуровневый интерфейс для посещения своих элементов.

Совместное использование методов Accept и группы методов Visit в паттерне Visitor обеспечивает *двойную диспетчеризацию* запросов, при которой характер запроса определяется двумя объектами: конкретным посетителем и конкретным элементом. Двойная диспетчеризация позволяет посетителю по-разному обрабатывать элементы различных классов.

Применение паттерна Visitor оправдано, если иерархия классов-элементов является стабильной (т. е. в нее редко добавляются новые классы) и при этом часто возникает необходимость в новых операциях, которые требуется по-разному выполнять для элементов различных типов.

Задание 1. Реализовать две иерархии классов, связанные с применением паттерна Visitor. Первая иерархия является иерархией классов-*элементов* и включает абстрактный класс Element, содержащий абстрактный метод Accept с параметром-ссылкой типа Visitor (не возвращает результата), и конкретные классы ConcreteElementA, ConcreteElementB, ConcreteElementC. Каждый конкретный класс содержит поле data; для класса ConcreteElementA оно целочисленное, для класса ConcreteElementB оно строковое, для класса ConcreteElementC оно является вещественным числом. Поле data инициализируется в конструкторе класса с помощью параметра data того же типа, что и инициализируемое поле. Конкретные классы-элементы A, B и C реализуют метод Accept(v), в котором для параметра v типа Visitor выполняется вызов соответствующего метода класса Visitor, определяемого типом класса-элемента: для элемента A это VisitConcreteElementA, для элемента B — VisitConcreteElementB, для элемента C — VisitConcreteElementC (параметром методов Visit является ссылка на объект, вызвавший метод Accept).

Кроме того, классы-элементы имеют методы для доступа на чтение и запись к полю data: метод GetData без параметров возвращает значение поля data, метод SetData с параметром newData изменяет значение поля data на значение параметра newData. Следует подчеркнуть, что методы GetData и SetData являются *специфическими* для каждого конкретного класса-элемента (в данном случае это аналоги специфических методов OperationA и OperationB, приведенных на диаграмме классов).

С иерархией классов-элементов также связан класс ObjectStructure. Поле struc этого класса является структурой (например, массивом) с элементами-ссылками на объекты типа Element (можно считать, что число элементов структуры struc не превосходит 10). Поле struc инициализируется в конструкторе с помощью параметра struc того же типа. Класс ObjectStructure содержит метод Accept(v) с параметром-ссылкой типа Visitor. Этот метод перебирает все элементы структуры struc и для каждого элемента вызывает его метод Accept с параметром v.

Вторая иерархия является иерархией классов-*посетителей*, связанных с ранее описанными конкретными классами. Она включает абстрактный класс Visitor и конкретные классы ConcreteVisitor1, ConcreteVisitor2 и ConcreteVisitor3. Класс Visitor содержит три абстрактных метода: VisitConcreteElementA(e), VisitConcreteElementB(e), VisitConcreteElementC(e). Эти методы не возвращают значений; их параметрами являются ссылки на соответствующие объекты-элементы (например, VisitConcreteElementA имеет параметр типа ConcreteElementA). Напомним, что именно эти методы должны вызываться в методе Accept каждого конкретного класса-элемента.

Конкретные классы-посетители 1, 2, 3 реализуют различные наборы операций, связанных с классами-элементами A, B, C, определяя методы VisitConcreteElementA(e), VisitConcreteElementB(e), VisitConcreteElementC(e). Класс ConcreteVisitor1 обеспечивает *вывод* поля data в окно задачника, вызывая в каждом из указанных методов соответствующую команду вывода для значения GetData элемента e. Класс ConcreteVisitor2 *преобразует* поле data различным образом для разных классов-элементов, используя метод

SetData элемента e: для элементов типа A он изменяет знак целого числа data на противоположный, для элементов типа B он изменяет порядок следования символов строки data на противоположный, для элементов типа C он изменяет любое ненулевое вещественное число data на обратное к нему (равное 1/data). Класс ConcreteVisitor3 определяет некоторую *общую характеристику* для всех однотипных элементов обрабатываемой структуры: для элементов типа A определяется сумма их целочисленных полей data, для элементов типа B находится строка, получаемая сцеплением всех строковых полей data, для элементов типа C находится произведение вещественных полей data. Для хранения полученных характеристик надо использовать в классе ConcreteVisitor3 поля resultA, resultB, resultC, а для доступа к ним — методы GetResultA, GetResultB, GetResultC. Конструкторы объектов-посетителей не имеют параметров и не выполняют дополнительных действий.

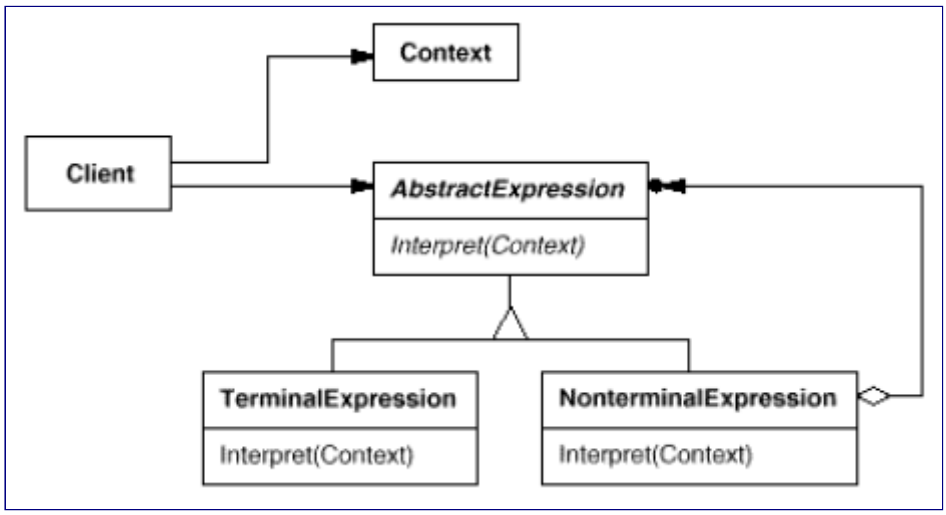
Дано целое число N (≤ 10) и N пар значений. Первое значение каждой пары представляет собой символ "A", "B" или "C"; он определяет тип создаваемого объекта-элемента (A, B или C). Второе значение каждой пары определяет поле data создаваемого элемента: для элемента типа A это целое число, для элемента типа B — строка, для элемента типа C — вещественное число.

Создать объект s типа ObjectStructure и поместить в него все исходные элементы. Также создать три объекта-посетителя v1, v2, v3 типа ConcreteVisitor1, ConcreteVisitor2, ConcreteVisitor3 соответственно. Вывести содержимое структуры s, используя вызов ее метода Accept(v1), после чего преобразовать это содержимое, используя вызов Accept(v2), и вывести преобразованное содержимое с помощью еще одного вызова Accept(v1). Затем вызвать метод Accept(v3) и вывести найденные в нем характеристики элементов структуры s с помощью методов GetResultA, GetResultB и GetResultC объекта v3.

```
public abstract class Element
{
    public abstract void Accept(Visitor v);
}

public class ConcreteElementA : Element
{
    // Add required fields and methods
    public override void Accept(Visitor v)
    {
        // Implement the method
    }
}

public class ConcreteElementB : Element
{
    // Add required fields and methods
    public override void Accept(Visitor v)
    {
        // Implement the method
    }
}

public class ConcreteElementC : Element
{
    // Add required fields and methods
    public override void Accept(Visitor v)
    {
        // Implement the method
    }
}

public class ObjectStructure
{
    Element[] struc;
    public ObjectStructure(Element[] struc)
    {
        // Implement the constructor
    }
```

```
        public void Accept(Visitor v)
        {
            foreach (var e in struc)
                e.Accept(v);
        }
    }

    public abstract class Visitor
    {
        public abstract void VisitConcreteElementA(ConcreteElementA e);
        public abstract void VisitConcreteElementB(ConcreteElementB e);
        public abstract void VisitConcreteElementC(ConcreteElementC e);
    }

    // Implement the ConcreteVisitor1, ConcreteVisitor2
    //   and ConcreteVisitor3 descendant classes
```

OOP3Behav16°.



Interpreter (**Интерпретатор**) — паттерн поведения.

Частота использования: низкая.

Назначение: для заданного языка определяет представление его грамматики, а также интерпретатор предложений этого языка.

Участники:

• *AbstractExpression* (*Абстрактное выражение*) — объявляет абстрактную операцию Interpret, общую для всех узлов в абстрактном синтаксическом дереве;

• *TerminalExpression* (*Терминальное выражение*) — реализует операцию Interpret для терминальных символов грамматики; необходим отдельный экземпляр для каждого терминального символа в предложении;

• *NonterminalExpression* (*Нетерминальное выражение*) — по одному такому классу требуется для каждого грамматического правила вида $R ::= R_1\ R_2\ \ldots\ R_n$; хранит переменные экземпляра типа AbstractExpression для каждого символа от R_1 до R_n; реализует операцию Interpret для нетерминальных символов грамматики (эта операция рекурсивно вызывает себя же для переменных, представляющих $R_1, \ldots, R_n$);

• *Context* (*Контекст*) — содержит информацию, глобальную по отношению к интерпретатору;

• *Client* (*Клиент*) — строит (или получает в готовом виде) абстрактное синтаксическое дерево разбора, представляющее отдельное предложение на языке с данной грамматикой (дерево составлено из экземпляров классов NonterminalExpression и TerminalExpression); вызывает операцию Interpret.

Поскольку разбор выражения не входит в задачу паттерна Interpreter (это может быть, например, задачей класса Client), в заданиях, связанных с этим паттерном, синтаксическое дерево разбора выражения предлагается в качестве одного из элементов исходных данных

(что позволяет, в частности, рассмотреть вариант операции Interpret, обеспечивающий восстановление исходного выражения по его дереву разбора).

Задание 1. Реализовать иерархию классов, которая определяет следующую грамматику арифметического выражения:

<expr> ::= <const> | <var> | <math>
<math> ::= (<expr><op><expr>)
<op> ::= + | − | * | /
<const> ::= <вещественное число>
<var> ::= <имя вещественной переменной>

Иерархия классов содержит абстрактный класс AbstractExpression, класс NontermMath (*математическая операция*), определяющий нетерминальное выражение, и два класса, определяющих терминальные выражения: TermConst (*константа*) и TermVar (*переменная*). Класс AbstractExpression содержит абстрактные методы InterpretA(cont), InterpretB(cont), InterpretC(cont), определяющие три варианта интерпретации выражения (параметр-ссылка cont имеет тип Context, описываемый далее). Методы InterpretA и InterpretB возвращают строковое значение, метод InterpretC — вещественное число. В каждом конкретном классе (NontermMath, TermConst и TermVar) требуется переопределить эти абстрактные методы.

Интерпретация A состоит в *восстановлении* строкового представления арифметического выражения, удовлетворяющего приведенной выше грамматике, по его синтаксическому дереву разбора. Интерпретация B состоит в *конструировании* строкового представления для эквивалентного выражения, записанного в *бесскобочном формате*, в котором терминальные выражения const и var остаются прежними, а нетерминальная операция math принимает вид <math> ::= <expr> <expr> <op> (между каждым элементом в правой части располагается пробел). Имена переменных для интерпретаций A и B берутся из контекста (экземпляра класса Context). Интерпретация C состоит в *вычислении* арифметического выражения, представленного синтаксическим деревом разбора; при этом значения переменных также берутся из контекста.

Примеры интерпретаций A, B, C для одного и того же синтаксического дерева: строка "(10.50−((var1+6.00)*a))", строка "10.50 var1 6.00 + a * −", число −9.5 (при условии, что контекст содержит переменные var1 = 4 и a = 2).

Класс Context должен содержать два набора элементов размера 10: строковый набор names с именами доступных переменных и набор вещественных чисел values со значениями соответствующих переменных (для хранения наборов можно использовать массив или другую структуру данных). Конструктор класса Context не имеет параметров; он заносит в набор names односимвольные имена переменных от a до j, а в набор values — значения 1.0. Класс Context включает три метода: SetVar(ind, name, value), GetName(ind), GetValue(ind). Параметр ind во всех методах определяет индекс обрабатываемой переменной (число от 0 до 9). Метод SetVar задает для переменной с индексом ind имя (строку name) и значение (вещественное число value). Метод GetName возвращает имя переменной с индексом ind, метод GetValue возвращает значение переменной с индексом ind. При реализации этих методов можно не проверять допустимость значений параметра ind, а также не контролировать возможную ошибочную ситуацию, когда два элемента набора name совпадают (т. е. когда две разные переменные имеют одинаковые имена).

Класс NontermMath содержит поля expr1 и expr2 — ссылки на AbstractExpression (первый и второй операнд математической операции соответственно) — и поле op символьного типа (знак операции). Поля инициализируются в конструкторе с помощью одноименных параметров.

Класс TermConst содержит вещественное поле value, задаваемое в конструкторе, имеющем параметр value. В методах InterpretA и InterpretB этого класса должно возвращаться строковое представление поля value с двумя дробными знаками и точкой в качестве десятичного разделителя.

Класс TermVar содержит целочисленное поле ind — индекс переменной в некотором объекте-контексте. Это поле инициализируется в конструкторе с помощью параметра ind.

Напомним, что контекст передается в качестве параметра во всех методах, выполняющих интерпретацию выражения (InterpretA, InterpretB и InterpretC).

Также определить класс Client, содержащий поля expr и cont — ссылки на AbstractExpression и Context, которые инициализируются в конструкторе с помощью соответствующих параметров. Класс Client включает три метода без параметров: InterpretA, InterpretB и InterpretC, в которых вызывается метод объекта expr с тем же именем и параметром cont и возвращается результат, полученный этим методом.

Дано целое число N (≤ 30) и N наборов значений, каждый из которых определяет один узел синтаксического дерева разбора. Последующие узлы могут содержать ссылки на предыдущие узлы, поэтому все узлы следует сохранять в структуре nodes (например, массиве) с элементами-ссылками типа AbstractExpression. Каждый набор, соответствующий узлу синтаксического дерева, начинается с символа "M", "C" или "V". Объекту класса NontermMath соответствует символ "M", за которым следуют три значения: индексы первого и второго операнда в уже заполненной части структуры nodes и символ операции (один из символов "+", "−", "*", "/"). Индексирование элементов структуры nodes ведется от 0. Объекту класса TermConst соответствует символ "C", за которым следует вещественное число — значение константы. Объекту класса TermVar соответствует символ "V", за которым следует индекс переменной в некотором контексте (целое число в диапазоне от 0 до 9).

Также дан набор значений, определяющих контекст: целое число M (≤ 10) и M наборов троек (ind, name, val), в котором ind определяет индекс переменной в контексте, name определяет имя переменной, а val — ее значение (для остальных переменных контекста сохраняются имена и значения по умолчанию).

Используя исходные данные, сформировать элементы синтаксического дерева разбора и сохранить их в структуре nodes, создать объект cont типа Context и настроить его содержимое. После этого создать объект cl типа Client, передав его конструктору последний элемент структуры nodes и объект cont. Для объекта cl вызвать методы InterpretA, InterpretB и InterpretC и вывести их возвращаемые значения.

```
public class Context
{
    // Add the constructor, required fields and methods
}

public abstract class AbstractExpression
{
    public abstract string InterpretA(Context cont);
    public abstract string InterpretB(Context cont);
    public abstract double InterpretC(Context cont);
}

// Implement the TermConst, TermVar
//   and NontermMath descendant classes

public class Client
{
    AbstractExpression expr;
    Context cont;
    public Client(AbstractExpression expr, Context cont)
    {
        this.expr = expr;
        this.cont = cont;
    }
    public string InterpretA()
    {
        return expr.InterpretA(cont);
    }
    public string InterpretB()
    {
        return expr.InterpretB(cont);
    }
    public double InterpretC()
    {
```

```
            return expr.InterpretC(cont);
        }
    }
```

OOP3Behav17°. **Interpreter** (**Интерпретатор**) — паттерн поведения.

Задание 2. Реализовать иерархию классов, которая определяет следующую грамматику строкового выражения:

<expr> : : = <concat> | <if> | <loop> | <str>
<concat> : : = <expr><expr> | <concat><expr>
<if> : : = (var?<expr>:<expr>)
<loop> : : = (var:<expr>)
<str> : : = <строка без символов "(", ")", "?", ":">
<var> : : = <имя целочисленной переменной>

Выражение concat возвращает конкатенацию нескольких выражений expr (двух или более); выражение if анализирует значение переменной var, и если var ≠ 0, то возвращает первое из указанных выражений expr, в противном случае возвращает второе из указанных выражений; выражение loop возвращает выражение expr, повторенное столько раз, каково значение переменной var (или пустую строку, если var ≤ 0).

Иерархия классов содержит абстрактный класс AbstractExpression, классы NontermConcat, NontermIf и NontermLoop, определяющие нетерминальные выражения concat, if, loop соответственно, и класс TermStr, определяющий терминальное выражение str. Класс AbstractExpression содержит два абстрактных метода InterpretA(cont) и InterpretB(cont), возвращающих строковое значение и определяющих два варианта интерпретации выражения (параметр-ссылка cont имеет тип Context, описываемый далее). В каждом конкретном классе (NontermConcat, NontermIf, NontermLoop и TermStr) требуется переопределить эти абстрактные методы.

Интерпретация A состоит в *восстановлении* строкового представления выражения, удовлетворяющего приведенной выше грамматике, по его синтаксическому дереву разбора; при этом имена переменных берутся из контекста (экземпляра класса Context). Интерпретация B состоит в *построении* конкретной строки по выражению, представленному синтаксическим деревом разбора; при этом значения переменных также берутся из контекста.

Примеры интерпретаций A и B для одного и того же синтаксического дерева разбора: строка "abc(var1?(n:x):dd)yz" и строка "abcxxxxyz" (при условии, что контекст содержит переменные var1 = 1 и n = 4).

Класс Context должен содержать два набора элементов размера 10: строковый набор names с именами доступных переменных и набор целых чисел values со значениями соответствующих переменных (для хранения наборов можно использовать массив или другую структуру данных). Конструктор класса Context не имеет параметров; он заносит в набор names односимвольные имена переменных от a до j, а в набор values — нулевые значения. Класс Context включает три метода: SetVar(ind, name, value), GetName(ind), GetValue(ind). Параметр ind во всех методах определяет индекс обрабатываемой переменной (число от 0 до 9). Метод SetVar задает для переменной с индексом ind имя (строку name) и значение (целое число value). Метод GetName возвращает имя переменной с индексом ind, метод GetValue возвращает значение переменной с индексом ind. При реализации этих методов можно не проверять допустимость значений параметра ind, а также не контролировать возможную ошибочную ситуацию, когда два элемента набора name совпадают (т.е. когда две разные переменные имеют одинаковые имена).

Класс NontermConcat содержит структуру exprs (например, массив) с элементами-ссылками типа AbstractExpression, которая инициализируется в конструкторе, имеющем соответствующий параметр-структуру. Можно считать, что выражение concat содержит не более 5 выражений expr. Класс NontermIf содержит поля expr1 и expr2 — ссылки на AbstractExpression (первое и второе выражение expr в правой части определения выражения if) — и целочисленное поле ind — индекс переменной var в некотором объекте-контексте. Класс NontermLoop содержит поле expr — ссылку на AbstractExpression (выражение expr в

правой части определения выражения loop) — и целочисленное поле ind — индекс переменной var в некотором объекте-контексте. Значения полей этих классов задаются в их конструкторах с помощью одноименных параметров.

Класс TermStr содержит строковое поле s, задаваемое в конструкторе с помощью строкового параметра. В методах InterpretA и InterpretB этого класса должно возвращаться значение поля s без каких-либо изменений.

Дано целое число N (≤ 30) и N наборов значений, каждый из которых определяет один узел синтаксического дерева разбора. Последующие узлы могут содержать ссылки на предыдущие, поэтому все узлы следует сохранять в структуре nodes (например, массиве) с элементами-ссылками типа AbstractExpression. Каждый набор, соответствующий узлу синтаксического дерева, начинается с символа "C", "I", "L" или "S". Объекту класса NontermConcat соответствует символ "C", за которым следует целое число K ($2 \le K \le 5$) и K индексов узлов из уже заполненной части структуры nodes (индексирование элементов структуры nodes ведется от 0); все узлы в указанном порядке должны входить в структуру exprs объекта NontermConcat. Объекту класса NontermIf соответствует символ "I", за которым следует индекс V переменной в некотором контексте (целое число в диапазоне от 0 до 9) и индексы двух узлов из уже заполненной части структуры nodes. Объекту класса NontermLoop соответствует символ "L", за которым следует индекс V переменной в некотором контексте и индекс некоторого узла из уже заполненной части структуры nodes. Наконец, объекту класса TermStr соответствует символ "S", за которым следует строка — значение выражения str.

Также даны три набора значений, определяющих три различных контекста. Определение каждого контекста содержит целое число M (≤ 10) и M наборов троек (ind, name, val), в которых ind определяет индекс переменной в контексте, name определяет имя переменной, а val — ее значение (для остальных переменных контекста сохраняются имена и значения по умолчанию).

Используя исходные данные, сформировать элементы синтаксического дерева разбора и сохранить их в структуре nodes, а также создать и настроить три объекта типа Context. Для последнего элемента структуры nodes вызвать методы InterpretA и InterpretB, указав в качестве параметра каждый из созданных контекстов, и вывести их возвращаемые значения (вначале выводятся значения, соответствующие первому контексту, затем второму, затем третьему).

```
public class Context
{
    // Add the constructor, required fields and methods
}

public abstract class AbstractExpression
{
    public abstract string InterpretA(Context cont);
    public abstract string InterpretB(Context cont);
}

// Implement the TermStr, NontermConcat, NontermIf
//   and NontermLoop descendant classes
```

Page generation date: 2022/4/26.