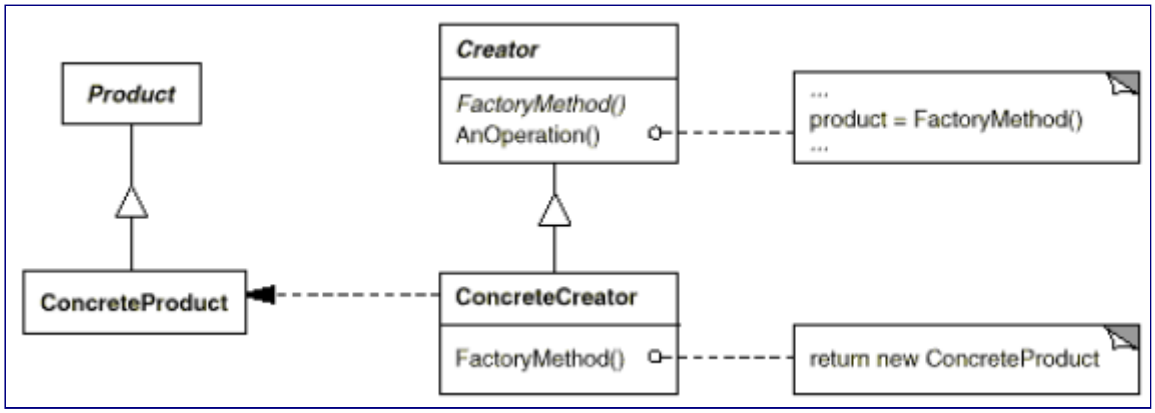
Порождающие паттерны

М. Э. Абрамян, 2021

Factory Method, Abstract Factory

OOP1Creat1°.



Factory Method (**Фабричный метод**) — порождающий паттерн.

Известен также под именем **Virtual Constructor** (**Виртуальный конструктор**).

Частота использования: высокая.

Назначение: определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Фабричный метод позволяет классу делегировать инстанцирование подклассам.

Участники:

• *Product* (*Продукт*) — определяет интерфейс объектов, создаваемых фабричным методом;

• *ConcreteProduct* (*Конкретный продукт*) — реализует интерфейс Product;

• *Creator* (*Создатель*) — объявляет фабричный метод, возвращающий объект типа Product; может также определять реализацию фабричного метода по умолчанию, возвращающую некоторый конкретный продукт; реализует методы, в которых используется объект Product, созданный фабричным методом;

• *ConcreteCreator* (*Конкретный создатель*) — замещает фабричный метод для создания конкретного продукта.

Задание 1. Реализовать две иерархии классов, в одну из которых входят абстрактный создатель Creator и два конкретных создателя ConcreteCreator1 и ConcreteCreator2, а в другую — абстрактный продукт Product и два конкретных продукта ConcreteProduct1 и ConcreteProduct2.

Абстрактный класс Product содержит два абстрактных метода, связанных с получением и преобразованием строки: метод GetInfo без параметров, возвращающий строку, и метод Transform без параметров, не возвращающий результат. Классы ConcreteProduct1 и ConcreteProduct2 содержат строковое поле info, которое инициализируется в конструкторе с помощью одноименного параметра, после чего в конструкторе класса ConcreteProduct1 поле info преобразуется к нижнему регистру, а в конструкторе класса ConcreteProduct2 — к верхнему. Метод GetInfo в каждом подклассе возвращает текущее значение поля info, а метод Transform преобразует это поле следующим образом: для ConcreteProduct1 он добавляет дополнительный пробел после каждого непробельного символа поля info (кроме его последнего символа), а для ConcreteProduct2 он добавляет два дополнительных символа * (звездочка) после каждого символа, отличного от звездочки (кроме последнего символа).

Абстрактный класс Creator содержит абстрактный фабричный метод FactoryMethod(info) со строковым параметром info, возвращающий ссылку на объект Product. Этот метод определяется в классах ConcreteCreator1 и ConcreteCreator2, причем фабричный метод класса ConcreteCreator1 создает объект типа ConcreteProduct1, а фабричный метод класса ConcreteCreator2 создает объект типа ConcreteProduct2; в любом случае конструктору создаваемого объекта передается параметр info фабричного метода.

В абстрактном классе Creator дополнительно определить метод AnOperation(info), который создает продукт с помощью фабричного метода, передавая ему параметр info, дважды вызывает метод Transform созданного продукта и с помощью его метода GetInfo возвращает полученный результат. Использование фабричного метода в методе AnOperation приводит к тому, что выполнение метода AnOperation в подклассах класса Creator дает различные результаты, зависящие от свойств создаваемых продуктов, причем такое поведение реализуется *без изменения кода* метода AnOperation.

Дано пять строк. Используя конкретных создателей 1 и 2, применить к каждой из данных строк метод AnOperation и вывести возвращаемый результат этого метода (вначале выводятся результаты для первой строки, затем для второй и т. д.).

```
public abstract class Product
{
    public abstract string GetInfo();
    public abstract void Transform();
}

// Implement the ConcreteProduct1
//   and ConcreteProduct2 descendant classes

public abstract class Creator
{
    protected abstract Product FactoryMethod(string info);
    public string AnOperation(string info)
    {
        Product p = FactoryMethod(info);
        p.Transform();
        p.Transform();
        return p.GetInfo();
    }
}

// Implement the ConcreteCreator1
//   and ConcreteCreator2 descendant classes;
//   the method AnOperation is not required in these classes
```

OOP1Creat2°. **Factory Method (Фабричный метод)** — порождающий паттерн.

Задание 2. Данное задание аналогично предыдущему (см. OOP1Creat1), однако в нем не используются абстрактные классы. Иерархию классов-*продуктов* представляет конкретный класс ConcreteProduct1 и его потомок ConcreteProduct2, иерархию классов-*создателей* представляет конкретный класс ConcreteCreator1 и его потомок ConcreteCreator2. Как и в предыдущем задании, классы-продукты обеспечивают хранение и получение строки (поле info и метод GetInfo, который возвращает значение поля info), а также ее преобразование (метод Transform). Поле Info инициализируется в конструкторе с помощью одноименного параметра, после чего конструктор класса ConcreteProduct1 преобразует поле info к нижнему регистру, а конструктор класса ConcreteProduct2 — к верхнему. Метод Transform для конкретного продукта 1 добавляет дополнительный символ * (звездочка) после каждого символа исходной строки info, отличного от звездочки (кроме последнего символа строки). Метод Transform для конкретного продукта 2 выполняет те же действия и дополнительно добавляет по два символа = (знак равенства) в начало и конец строки.

В классах ConcreteProduct1 и ConcreteProduct2 определить фабричный метод FactoryMethod(info) со строковым параметром, возвращающий ссылку на объект ConcreteProduct1. Фабричный метод класса ConcreteCreator1 создает объект типа ConcreteProduct1, а фабричный метод класса ConcreteCreator2 создает объект типа ConcreteProduct2; в любом случае конструктору создаваемого объекта передается параметр info фабричного метода. В классе ConcreteCreator1 также определить метод AnOperation(info), который создает продукт с помощью фабричного метода, передавая ему параметр info, дважды вызывает метод Transform созданного продукта и с помощью его метода GetInfo возвращает полученный результат.

Дано пять строк. Используя конкретных создателей 1 и 2, применить к каждой из данных строк метод AnOperation и вывести возвращаемый результат этого метода (вначале выводятся результаты для первой строки, затем для второй и т. д.).

Указание. Класс-потомок ConcreteProduct2 модифицирует поведение класса-предка ConcreteProduct1. При определении конструктора класса ConcreteProduct2 используйте выражение : base(info) после его заголовка (для вызова конструктора предка), при модификации метода Transform класса ConcreteProduct2 используйте вызов base.Transform(). Класс-потомок ConcreteCreator2 модифицирует поведение класса-предка ConcreteCreator1, подменяя определенный в предке способ создания продукта.

```
public class ConcreteProduct1
{
    protected string info;
    public string GetInfo()
    {
        return info;
    }
    public ConcreteProduct1(string info)
    {
        // Implement the constructor
    }
    public virtual void Transform()
    {
        // Implement the method
    }
}

// Implement the ConcreteProduct2 descendant class

public class ConcreteCreator1
{
    protected virtual ConcreteProduct1 FactoryMethod(string info)
    {
        return new ConcreteProduct1(info);
    }
    public string AnOperation(string info)
    {
        ConcreteProduct1 p = FactoryMethod(info);
        p.Transform();
        p.Transform();
        return p.GetInfo();
    }
}

// Implement the ConcreteCreator2 descendant class
```

OOP1Creat3°. **Factory Method (Фабричный метод)** — порождающий паттерн.

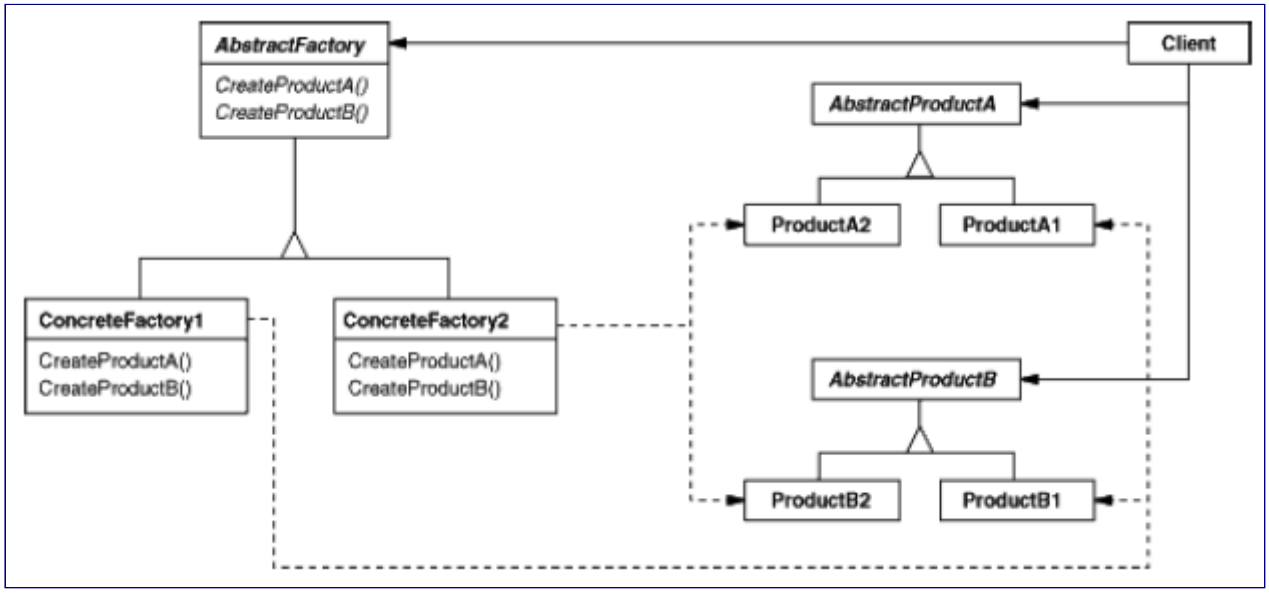
Задание 3. Реализовать иерархию классов-животных с абстрактным предком Animal, содержащим метод GetInfo, который возвращает название класса, и шестью конкретными потомками: Cobra, Python, Anaconda (змеи, snakes), Gorilla, Orangutan, Chimpanzee (человекообразные обезьяны, apes). Реализовать иерархию классов-создателей с абстрактным предком AnimalCreator и конкретными потомками SnakeCreator и ApeCreator. Фабричный метод CreateAnimal(n) этих классов принимает один целочисленный параметр n и возвращает объект типа Animal. Для конкретных классов SnakeCreator и ApeCreator параметр n метода CreateAnimal определяет номер создаваемого животного (1, 2 или 3) в соответствующей группе (змей или обезьян). В классе AnimalCreator дополнительно определить метод GetZoo с параметром-массивом id целых чисел (предполагается, что элементы массива id всегда имеют значения в диапазоне от 1 до 3). Метод GetZoo возвращает массив объектов Animal того же размера, что и массив id; каждый элемент результирующего массива определяется с помощью фабричного метода с параметром, равным значению соответствующего элемента массива id.

Дан набор из 5 чисел, каждое из которых находится в диапазоне от 1 до 3. Используя метод GetZoo для создателей SnakeCreator и ApeCreator, получить наборы змей и обезьян размера 5 и вывести названия животных из каждого набора.

```
public abstract class Animal
{
    public abstract string GetInfo();
}
```

```
// Implement the Cobra, Python, Anaconda, Gorilla,
//   Orangutan and Chimpanzee descendant classes

public abstract class AnimalCreator
{
    protected abstract Animal CreateAnimal(int n);
    public Animal[] GetZoo(int[] id)
    {
        Animal[] zoo = new Animal[id.Length];
        for (int i = 0; i < zoo.Length; i++)
            zoo[i] = CreateAnimal(id[i]);
        return zoo;
    }
}

// Implement the SnakeCreator and ApeCreator descendant classes
```

OOP1Creat4°.



Abstract Factory (**Абстрактная фабрика**) — порождающий паттерн.

Известен также под именем **Kit** (**Инструментарий**).

Частота использования: высокая.

Назначение: предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя конкретные классы этих объектов. Методы абстрактной фабрики обычно реализуются как *фабричные методы* (см. OOP1Creat1).

Участники:

• *AbstractFactory* (*Абстрактная фабрика*) — объявляет интерфейс для операций, создающих абстрактные объекты-продукты;

• *ConcreteFactory* (*Конкретная фабрика*) — реализует операции, создающие конкретные объекты-продукты;

• *AbstractProduct* (*Абстрактный продукт*) — объявляет интерфейс для типа объекта-продукта;

• *ConcreteProduct* (*Конкретный продукт*) — определяет объект-продукт, создаваемый соответствующей конкретной фабрикой; реализует интерфейс AbstractProduct;

• *Client* (*Клиент*) — пользуется исключительно интерфейсами, которые объявлены в классах AbstractFactory и AbstractProduct.

Задание 1. Реализовать три иерархии классов, в одну из которых входят абстрактная фабрика AbstractFactory и две конкретные фабрики ConcreteFactory1 и ConcreteFactory2, в другую — абстрактный продукт (типа A) AbstractProductA и два его потомка ProductA1 и ProductA2, а в третью — абстрактный продукт (типа B) AbstractProductB и два его потомка ProductB1 и ProductB2. Все фабрики включают методы CreateProductA и CreateProductB, конкретные фабрики 1 и 2 возвращают конкретные продукты с соответствующим номером

(фабрика 1 возвращает продукты ProductA1 и ProductB1, фабрика 2 — продукты ProductA2 и ProductB2).

Все классы-продукты имеют метод GetInfo, возвращающий строковое значение. Кроме того, в продукте первого типа определен метод A без параметров, а в продукте второго типа — метод B с параметром типа AbstractProductA (методы не возвращают значений). Конкретные продукты содержат строковое поле info, которое инициализируется в конструкторе с помощью его параметра *целого типа* (в поле info записывается строковое представление целочисленного параметра конструктора). Метод GetInfo конкретных классов-продуктов возвращает текущее значение поля info.

Для класса-продукта ProductA1 метод A переводит поле info в целое число, удваивает его и сохраняет строковое представления результата в поле info; для класса-продукта ProductA2 метод A просто удваивает строку info. Для класса-продукта ProductB1 вызов objB.B(objA) преобразует поля objB.info и objA.info в целые числа, складывает их и сохраняет строковое представление результата в поле objB.info; для класса-продукта ProductB2 вызов objB.B(objA) находит сумму строк objB.info + objA.info и сохраняет результат в поле objB.info.

Дано целое число Nf, которое может быть равно 1 или 2, целые числа Na и Nb и строка S, содержащая только символы A и B. Описать ссылочные переменные f типа AbstractFactory, pa типа AbstractProductA и pb типа AbstractProductB. Если число Nf равно 1, то связать f с конкретной фабрикой 1, если Nf равно 2, то связать f с конкретной фабрикой 2. Используя фабричные методы созданной фабрики, создать конкретные продукты типа A и B, инициализировав их данными числами Na и Nb соответственно, и связать их с переменными pa и pb. Затем для созданных продуктов pa и pb выполнить методы A и B в порядке, указанном в исходной строке S. При этом метод A должен вызываться для продукта pa, а метод B — для продукта pb, причем параметром метода B должен быть продукт pa. Используя методы GetInfo, вывести итоговые значения объектов-продуктов pa и pb (в указанном порядке).

Примечание. При выполнении задания используются только ссылки на абстрактные классы, а также только методы, определенные в абстрактных классах (за исключением конструктора создаваемой конкретной фабрики), что и составляет суть паттерна "Абстрактная фабрика".

```
public abstract class AbstractProductA
{
    public abstract void A();
    public abstract string GetInfo();
}

// Implement the ProductA1 and ProductA2 descendant classes

public abstract class AbstractProductB
{
    public abstract void B(AbstractProductA objA);
    public abstract string GetInfo();
}

// Implement the ProductB1 and ProductB2 descendant classes

public abstract class AbstractFactory
{
    public abstract AbstractProductA CreateProductA(int info);
    public abstract AbstractProductB CreateProductB(int info);
}

// Implement the ConcreteFactory1
//   and ConcreteFactory2 descendant classes
```

OOP1Creat5°. **Abstract Factory** (**Абстрактная фабрика**) — порождающий паттерн.

Задание 2. Реализовать иерархию классов, определяющих два вида элементов управления (controls): кнопки (buttons) и метки (labels). Абстрактные классы AbstractButton и AbstractLabel содержат метод GetControl, возвращающий строковое представление соответствующего элемента управления. Конкретные классы Button1, Button2, Label1,

Label2 включают конструктор со строковым параметром text, который определяет текст, отображаемый на элементе управления (текст хранится в поле text). Конкретные классы отличаются видом строкового представления. Для Button1 и Label1 (первый тип представления) текст отображается заглавными буквами, кнопки обрамляются квадратными скобками (например, [CAPTION]), метки обрамляются символами = (например, =MESSAGE=). Для Button2 и Label2 (второй тип представления) текст отображается строчными (маленькими) буквами, кнопки обрамляются угловыми скобками (например, <caption>), метки обрамляются двойными кавычками (например, "message").

Реализовать иерархию классов ControlFactory (абстрактная фабрика), Factory1 и Factory2 (конкретные фабрики). Каждый класс содержит два метода: CreateButton(text) и CreateLabel(text). Для ControlFactory эти методы являются абстрактными, для конкретных фабрик они возвращают кнопку и метку соответствующего типа (первого или второго соответственно).

Также реализовать класс Client, предназначенный для формирования набора элементов управления. Конструктор данного класса принимает параметр f типа ControlFactory, который в дальнейшем используется для генерации элементов управления требуемого типа. Класс Client включает метод AddButton(text) для добавления в набор новой кнопки, метод AddLabel(text) для добавления в набор новой метки и метод GetControls, возвращающий текстовое представление полученного набора элементов управления. В текстовом представлении каждый последующий элемент отделяется от предыдущего одним пробелом.

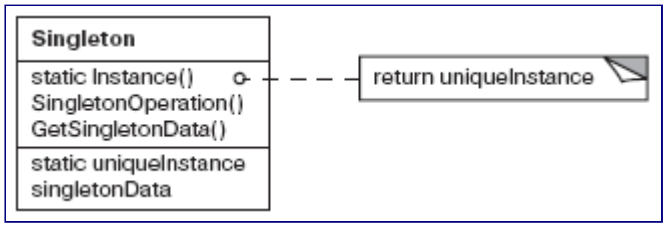
Дано целое число N (≤ 6) и набор из N строк. Каждая строка начинается либо с символа B (признак кнопки), либо с символа L (признак метки). Затем идет пробел и текст соответствующего элемента управления. Используя два экземпляра класса Client, сформировать и вывести два варианта текстового представления указанного набора элементов управления. Вначале выводится представление первого типа, затем второго.

```
public abstract class AbstractButton
{
    public abstract string GetControl();
}

// Implement the Button1 and Button2 descendant classes

public abstract class AbstractLabel
{
    public abstract string GetControl();
}

// Implement the Label1 and Label2 descendant classes

public abstract class ControlFactory
{
    public abstract AbstractButton CreateButton(string text);
    public abstract AbstractLabel CreateLabel(string text);
}

// Implement the Factory1 and Factory2 descendant classes

public class Client
{
    // Add required fields
    public Client(ControlFactory f)
    {
        // Implement the constructor
    }
    public void AddButton(string text)
    {
        // Implement the method
    }
    public void AddLabel(string text)
    {
        // Implement the method
    }
    public string GetControls()
```

```
        {
            return "";
            // Remove the previous statement and implement the method
        }
    }
```

Singleton, Prototype, Builder

OOP1Creat6°.



Singleton (**Одиночка**) — порождающий паттерн.

Частота использования: выше средней.

Назначение: гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Участники:

• *Singleton* (*Одиночка*) — определяет операцию Instance, которая позволяет клиентам получать доступ к единственному экземпляру (операция Instance обычно оформляется в виде *статического*, т. е. классового, метода); может обеспечивать *отложенную инициализацию* данного экземпляра.

Чтобы проиллюстрировать особенности паттерна Singleton, в задании предлагается реализовать не только его стандартный вариант, но и основанные на той же идее варианты, допускающие использование *ограниченного количества* экземпляров.

Задание 1. Реализовать иерархию классов, включающую абстрактный базовый класс BaseClass и классы-потомки Singleton, Doubleton и Tenton, обеспечивающие создание ограниченного количества своих экземпляров.

Класс BaseClass включает целочисленное поле data и два связанных с ним метода: метод IncData(increment) увеличивает значение data на величину целочисленного параметра increment, метод GetData без параметров возвращает текущее значение поля data. Поле data инициализируется нулевым значением.

Класс Singleton реализует стандартный паттерн "Одиночка". Он включает статическое поле uniqueInstance — ссылку на тип Singleton (инициализируется нулевой ссылкой), закрытый конструктор без параметров, не выполняющий дополнительных действий, и статический метод Instance (без параметров, возвращает ссылку на тип Singleton). Метод Instance выполняет следующие действия: если поле uniqueInstance является нулевой ссылкой, то метод создает объект Singleton, помещает ссылку на него в поле uniqueInstance и возвращает эту ссылку как результат своей работы; если поле uniqueInstance уже содержит ссылку на объект Singleton, то метод Instance просто возвращает эту ссылку. Дополнительно реализовать статический метод InstanceCount (без параметров), который возвращает 0, если поле uniqueInstance содержит нулевую ссылку, и 1 в противном случае.

Классы Doubleton и Tenton реализуют вариант паттерна, допускающий использование *не более двух* и *не более десяти* экземпляров соответственно.

Класс Doubleton включает статический массив instances из двух элементов-ссылок типа Doubleton (элементы инициализируются нулевыми значениями), закрытый конструктор и статические методы Instance1 и Instance2, которые выполняют действия, аналогичные действиям метода Instance класса Singleton, но обращаются соответственно к элементам массива instances с индексами 0 и 1.

Класс Tenton отличается от класса Doubleton тем, что его статический массив instances содержит 10 элементов-ссылок типа Tenton, а вместо двух методов Instance1 и Instance2 он включает статический метод Instance(index) с целочисленным параметром, определяющим индекс элемента массива instances, к которому обращается данный метод. Если параметр

находится вне диапазона 0–9, то метод Instance может возвращать нулевую ссылку или возбуждать исключение (при выполнении задания такая ситуация не будет возникать).

В классах Doubleton и Tenton дополнительно реализовать статический метод InstanceCount (без параметров), который возвращает количество элементов массива instances, не являющихся нулевыми ссылками.

Дано целое число N (≤ 10) и набор из N строк, которые могут принимать значения "S", "D1", "D2", "T0", "T1", …, "T9". Создать массив b из N элементов — ссылок на BaseClass и инициализировать его элементы экземплярами классов Singleton, Doubleton, Tenton, используя следующие варианты статических методов в зависимости от значения соответствующей строки из исходного набора: для строки "S" используется метод Instance класса Singleton; для строк "D1", "D2" — соответственно методы Instance1 и Instance2 класса Doubleton; для строк "T0", "T1", …, "T9" — метод Instance(index) класса Tenton с параметром index, соответствующим цифре, указанной в строке.

После создания всех элементов массива b вывести значения метода InstanceCount для классов Singleton, Doubleton, Tenton в указанном порядке.

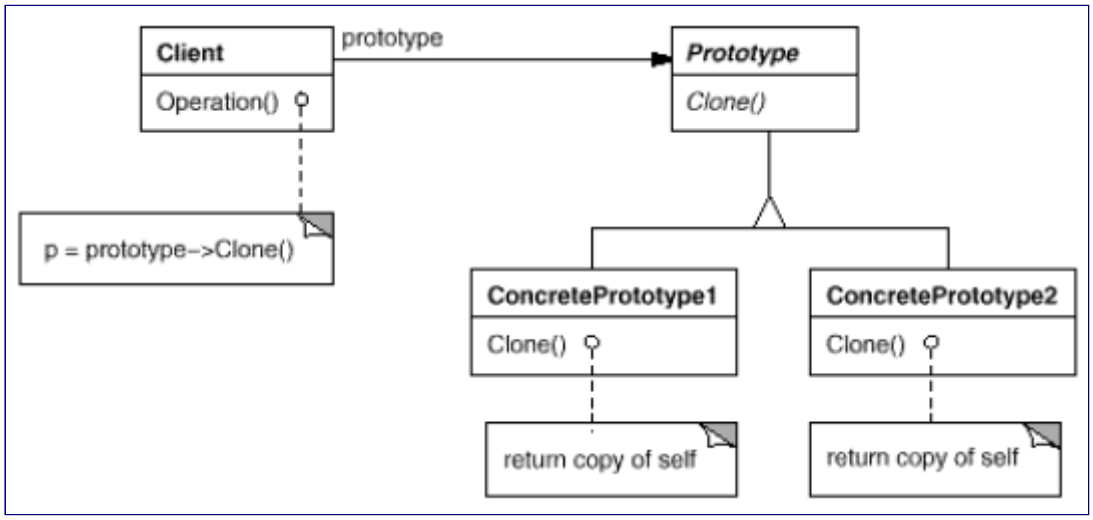
Также дано целое число K (≤ 20) и набор из K пар целых чисел (index, increment), в котором первое число находится в диапазоне от 0 до N − 1 и определяет индекс элемента в массиве b, а второе число определяет параметр метода IncData(increment), который надо вызвать для элемента b[index]. После вызова всех требуемых методов IncData вывести итоговые значения поля data для всех объектов массива b, используя метод GetData.

Указание. Для языка C# задачник Programming Taskbook выполняет тестирование при *однократном* запуске программы путем *многократного* вызова ее функции Solve. Так как между вызовами функции Solve содержимое статических полей классов сохраняется, в случае данного задания это приведет к ошибочным результатам, начиная со *второго* тестового испытания. Чтобы избежать сообщения об ошибке, следует реализовать в классах Singleton, Doubleton и Tenton вспомогательный статический метод Reset, который обнуляет все статические ссылки (uniqueInstance для класса Singleton, элементы массива instances для классов Doubleton и Tenton), и вызывать методы Reset для этих классов после вывода всех результирующих данных.

```
public abstract class BaseClass
{
    int data;
    public void IncData(int increment)
    {
        data += increment;
    }
    public int GetData()
    {
        return data;
    }
}

public class Singleton : BaseClass
{
    static Singleton uniqueInstance;
    Singleton() {}
    public static void Reset()
    {
        uniqueInstance = null;
    }
    // Complete the implementation of the class
}

public class Doubleton : BaseClass
{
    static Doubleton[] instances = new Doubleton[2];
    Doubleton() {}
    public static void Reset()
    {
        instances[0] = instances[1] = null;
    }
```

```
        // Complete the implementation of the class
    }

    public class Tenton : BaseClass
    {
        static Tenton[] instances = new Tenton[10];
        Tenton() {}
        public static void Reset()
        {
            for (int i = 0; i < instances.Length; i++)
                instances[i] = null;
        }
        // Complete the implementation of the class
    }
```

OOP1Creat7°.



Prototype (**Прототип**) — порождающий паттерн.

Частота использования: средняя.

Назначение: задает виды создаваемых объектов с помощью экземпляра-прототипа и создает новые объекты путем копирования ("клонирования") этого прототипа.

Участники:

• *Prototype* (*Прототип*) — объявляет интерфейс для клонирования самого себя;

• *ConcretePrototype* (*Конкретный прототип*) — реализует операцию клонирования себя;

• *Client* (*Клиент*) — создает новый объект, обращаясь к прототипу с запросом клонировать себя.

Задание 1. Реализовать иерархию классов, которая содержит абстрактный прототип Prototype и два конкретных прототипа ConcretePrototype1 и ConcretePrototype2. Все классы включают метод Clone без параметров, возвращающий копию объекта, вызвавшего данный метод, а также методы GetInfo и ChangeId. Метод GetInfo без параметров возвращает строку, метод ChangeId имеет целочисленный параметр id и не возвращает значений. В классе Prototype методы Clone, GetInfo и ChangeId являются абстрактными. Конкретные прототипы содержат строковое поле data и целочисленное поле id, которые инициализируются соответствующими параметрами конструктора. Метод GetInfo для конкретных прототипов возвращает строку, содержащую краткое имя типа (CP1 для типа ConcretePrototype1 и CP2 для типа ConcretePrototype2) и значения полей data и id (части описания разделяются символом "=", например, "CP1=TEXT=34"). Метод ChangeId изменяет значение поля id. При реализации метода Clone можно использовать специальные средства стандартной библиотеки или обычный вызов конструктора.

Также реализовать класс Client, предназначенный для работы с группой объектов типа ConcretePrototype1 или ConcretePrototype2. Конструктор класса Client имеет параметр-ссылку типа Prototype, определяющий прототип объектов, включаемых в группу (прототип в группу не входит и сохраняется в специальном поле prot; для хранения группы объектов удобно использовать динамическую структуру). Класс Client также содержит методы AddObject(id) и GetObjects. Метод AddObject добавляет в набор новый объект, получаемый

путем клонирования прототипа и последующего изменения поля id полученного объекта в соответствии со значением параметра метода AddObject. Метод GetObjects без параметров возвращает строку с описанием всех объектов группы (описания разделяются пробелом). Все объекты создаваемой группы имеют одно и то же поле data и различные поля id (задаваемые в методе AddObject).

Дана строка S, целое число N (≤ 10) и набор из N целых чисел. С помощью двух объектов класса Client сформировать два набора из N объектов типа ConcretePrototype1 (для первого объекта Client) и ConcretePrototype2 (для второго объекта Client). Все созданные объекты должны иметь одинаковые поля data, равные строке S, и значения полей id, взятые из исходного набора целых чисел. Используя метод GetObjects, вывести строковые описания каждого из полученных наборов объектов.

Примечание. В реальной ситуации подход, основанный на паттерне "Прототип" будет эффективным, если объем действий по обычному созданию объекта превосходит объем действий по его клонированию и последующей настройке некоторых свойств.

```
public abstract class Prototype
{
    public abstract Prototype Clone();
    public abstract void ChangeId(int id);
    public abstract string GetInfo();
}

// Implement the ConcretePrototype1
//   and ConcretePrototype2 descendant classes

public class Client
{
    // Add required fields
    public Client(Prototype p)
    {
        // Implement the constructor
    }
    public void AddObject(int id)
    {
        // Implement the method
    }
    public string GetObjects()
    {
        return "";
        // Remove the previous statement and implement the method
    }
}
```

OOP1Creat8°. **Prototype** (**Прототип**) — порождающий паттерн.

Задание 2. Реализовать иерархию классов, связанных с графическими примитивами: AbstractGraphic (абстрактный предок), Ellip, Line и Rect (конкретные примитивы). Классы содержат метод Clone без параметров, возвращающий копию объекта, вызвавшего данный метод, а также метод ChangeLocation(x1, y1, x2, y2) с целочисленными параметрами x1, y1, x2, y2, не возвращающий значений, и метод Draw без параметров, возвращающий строку. В классе AbstractGraphic методы Clone, ChangeLocation и Draw являются абстрактными. Классы конкретных примитивов содержат целочисленные поля x1, y1, x2, y2; по умолчанию они инициализируются нулями. Метод ChangeLocation изменяет эти поля, а метод Draw возвращает строку, содержащую имя класса и текущие значения полей без пробелов (например, "Line(1,3,-1,5)"). При реализации метода Clone можно использовать специальные средства стандартной библиотеки или обычный вызов конструктора.

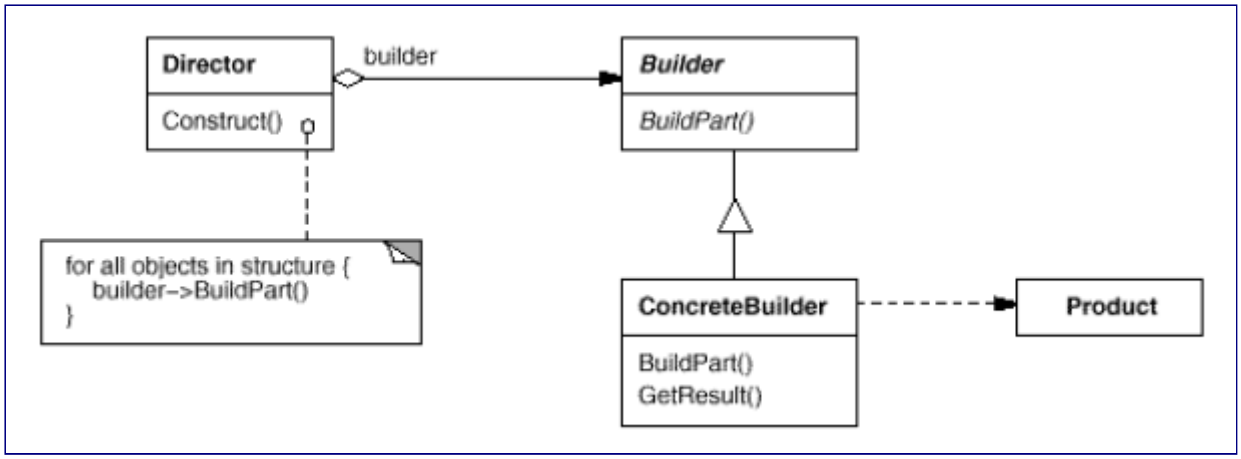
Также реализовать класс GraphEditor, предназначенный для работы с графическими объектами. В конструкторе класса GraphEditor передаются два ссылочных параметра p1, p2 типа AbstractGraphic, определяющих прототипы создаваемых объектов-примитивов. Для хранения прототипов используется массив из двух элементов, для хранения созданного набора графических примитивов удобно использовать динамическую структуру. Класс GraphEditor включает два метода. Метод AddGraphic(np, x1, y1, x2, y2) добавляет в набор графических примитивов объект, созданный на основе прототипа с номером np (1 или 2) и

устанавливает для него указанные координаты. Метод DrawAll без параметров возвращает строковое описание всех добавленных графических объектов, используя их метод Draw (описания объектов разделяются пробелом).

Дана двухсимвольная строка P, содержащая две различные буквы из набора E, L, R. Также дано целое число N (≤ 5) и набор из N пятерок целых чисел вида (np, x1, y1, x2, y2), где np принимает значение 1 или 2, а остальные числа являются произвольными. Создать объект GraphEditor, инициализировав его двумя прототипами, которые соответствуют символам строки P (E — Ellip, L — Line, R — Rect); порядок прототипов определяется порядком символов в строке P. Добавить в набор графических примитивов N объектов, используя вызовы метода AddGraphic с параметрами, определяемыми пятерками данных чисел, и вывести полученный набор примитивов методом DrawAll.

```
public abstract class AbstractGraphic
{
    public abstract AbstractGraphic Clone();
    public abstract void ChangeLocation(int x1, int y1, int x2, int y2);
    public abstract string Draw();
}

// Implement the Ellip, Line and Rect descendant classes

public class GraphEditor
{
    // Add required fields
    public GraphEditor(AbstractGraphic p1, AbstractGraphic p2)
    {
        // Implement the constructor
    }
    public void AddGraphic(int np, int x1, int y1, int x2, int y2)
    {
        // Implement the method
    }
    public string DrawAll()
    {
        return "";
        // Remove the previous statement and implement the method
    }
}
```

OOP1Creat9°.



Builder (**Строитель**) — порождающий паттерн.

Частота использования: ниже средней.

Назначение: отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления.

Участники:

• *Builder* (*Строитель*) — задает абстрактный интерфейс для создания частей объекта Product;

• *ConcreteBuilder* (*Конкретный строитель*) — конструирует и собирает вместе части продукта посредством реализации интерфейса Builder, предоставляет интерфейс для доступа к продукту;

• *Director* (*Распорядитель*) — конструирует объект Product, пользуясь интерфейсом Builder;

• *Product* (*Продукт*) — представляет сложный конструируемый объект.

Задание 1. Реализовать иерархию классов-строителей, конструирующих продукты-строки. Иерархия включает абстрактный класс Builder, который предоставляет интерфейс для инициализации продукта (BuildStart) и создания трех его фрагментов (BuildPartA, BuildPartB, BuildPartC), и конкретные классы ConcreteBuilder1 и ConcreteBuilder2, которые определяют конкретные способы конструирования. Методы BuildStart, BuildPartA, BuildPartB, BuildPartC не имеют параметров и не возвращают значений. В абстрактном классе Builder подобные методы обычно не выполняют никаких действий, хотя и не являются абстрактными; это позволяет конкретному классу-строителю не переопределять некоторые из них, если его устраивает поведение по умолчанию. Кроме того, в классе Builder определен абстрактный метод GetResult без параметров, возвращающий строку-продукт.

Конкретные классы ConcreteBuilder1 и ConcreteBuilder2 содержат строковое поле product; их метод GetResult возвращает текущее значение поля product. В конструкторе этих классов поле product инициализируется пустой строкой, это же действие выполняет и метод BuildStart. Каждый из методов BuildPartA, BuildPartB, BuildPartC добавляет к строке product новый текстовый фрагмент; для класса ConcreteBuilder1 фрагменты A, B, C представляют собой строки "-1-", "-2-" и "-3-", а для класса ConcreteBuilder2 — строки "=*=", "=**=" и "=***=".

Также определить класс Director, содержащий поле b — ссылку на объект типа Builder (поле b инициализируется в конструкторе с использованием параметра конструктора того же типа) и два метода: Construct(templat) и GetResult. Метод GetResult не имеет параметров и возвращает значение метода GetResult объекта b, т. е. построенный продукт. Метод Construct обеспечивает построение продукта; его строковый параметр templat определяет план строительства. В данном случае план определяется последовательностью символов "A", "B", "C", каждый из которых соответствует фрагменту A, B, C, добавляемому к конструируемой строке в указанном порядке. Строка templat может содержать символы, отличные от "A", "B", "C"; подобные символы игнорируются. В начале своей работы метод Construct должен вызвать метод BuildStart.

Даны пять строк, каждая из которых содержит план строительства. Создать два экземпляра d1 и d2 класса Director, передав первому экземпляру объект типа ConcreteBuilder1, а второму — объект типа ConcreteBuilder2. Вызвать метод Construct объектов d1 и d2 для каждой из исходных строк и вывести полученные результаты, используя метод GetResult (вначале выводятся результаты для первой исходной строки, затем для второй и т. д.).

```
public abstract class Builder
{
    public virtual void BuildStart() {}
    public virtual void BuildPartA() {}
    public virtual void BuildPartB() {}
    public virtual void BuildPartC() {}
    public abstract string GetResult();
}

// Implement the ConcreteBuilder1
//   and ConcreteBuilder2 descendant classes

public class Director
{
    Builder b;
    public Director(Builder b)
    {
        this.b = b;
    }
    public string GetResult()
    {
```

```
            return b.GetResult();
        }
        public void Construct(string templat)
        {
            // Implement the method
        }
    }
```

OOP1Creat10°. **Builder** (**Строитель**) — паттерн, порождающий объекты.

Задание 2. Реализовать систему классов, позволяющую по строковым описаниям генерировать идентификаторы, которые удовлетворяют соглашениям различных языков программирования. Каждое строковое описание представляет собой одно или более слов, разделенных одним или несколькими пробелами; начальные и конечные пробелы отсутствуют, регистр букв является произвольным.

Абстрактный класс Builder содержит методы для конструирования первого символа идентификатора (BuildStart), первого символа каждого последующего слова (BuildFirstChar), последующих символов слов (BuildNextChar) и символов-разделителей между словами (BuildFirstSpace). Методы BuildStart, BuildFirstChar и BuildNextChar имеют символьный параметр, используемый при конструировании (возможно, после изменения его регистра); метод BuildFirstSpace не имеет параметров. Все эти методы не возвращают значений; в классе Builder они не выполняют никаких действий. Кроме того, класс Builder содержит абстрактный метод GetResult без параметров, возвращающий строковое значение.

Конкретные классы BuilderPascal, BuilderPython, BuilderC содержат строковое поле product, которое инициализируется пустой строкой в конструкторе, обновляется в методе BuildStart и дополняется в методах BuildFirstChar, BuildNextChar и, возможно, в методе BuildFirstSpace. Метод GetResult возвращает значение поля product. При определении остальных методов конкретных строителей следует учитывать правила формирования идентификаторов для конкретных языков программирования:

• для языка Pascal идентификатор должен начинаться со строчной (маленькой) буквы, последующие слова — с заглавной буквы, все прочие буквы являются строчными, пробелы игнорируются;

• для языка Python все буквы являются строчными, а между словами добавляется символ подчеркивания;

• для языка C все буквы являются строчными, а пробелы игнорируются.

Из перечисленных правил следует, в частности, что для классов BuilderPascal и BuilderC не требуется переопределять метод BuildFirstSpace.

Также определить класс Director, содержащий ссылочное поле b типа Builder (поле b инициализируется в конструкторе с использованием параметра конструктора того же типа), и два метода: Construct(templat) и GetResult. Метод GetResult не имеет параметров и возвращает значение метода GetResult объекта b. Метод Construct обеспечивает построение продукта; его строковый параметр templat содержит строковое описание, на основании которого должен конструироваться идентификатор по правилам, реализованным в строителе b. В начале работы метод Construct должен вызвать метод BuildStart с параметром — первым символом строки templat. При анализе строки templat необходимо различать первый пробел в последовательности пробелов (для которого надо вызывать метод BuildFirstSpace) и последующие пробелы (которые должны игнорироваться).

Даны пять строк, содержащих строковые описания, которые удовлетворяют условиям, перечисленным в начале формулировки задания. Создать три объекта-распорядителя Director, связанных со строителями BuilderPascal, BuilderPython, BuilderC. Используя методы Construct и GetResult каждого из созданных распорядителей, получить по каждой из исходных строк идентификаторы на языке Pascal, Python и C и вывести их в указанном порядке (вначале выводятся идентификаторы, полученные на основе первой строки, затем на основе второй и т. д.).

```
public abstract class Builder
{
    public virtual void BuildStart(char c) {}
    public virtual void BuildFirstChar(char c) {}
    public virtual void BuildNextChar(char c) {}
```

```
        public virtual void BuildFirstSpace() {}
        public abstract string GetResult();
    }

    // Implement the BuilderPascal, BuilderPyhton
    //   and BuilderC descendant classes

    public class Director
    {
        Builder b;
        public Director(Builder b)
        {
            this.b = b;
        }
        public string GetResult()
        {
            return b.GetResult();
        }
        public void Construct(string templat)
        {
            b.BuildStart(templat[0]);
            // Complete the implementation of the method
        }
    }
```

Page generation date: 2022/4/26.