LSP: Liskov Substitution Principle

(принцип замещения Лисков)

Функции, которые используют ссылки на базовые классы, должны иметь возможность использовать объекты производных классов, не зная об этом.

Содержание

- Наследование
- Пример иерархии
- Контракты
- Паттерн Одиночка

Критерий для оценки качества принимаемых решений при построении иерархий наследования.

Впервые этот принцип был упомянут Барбарой Лисков в 1987 году на научной конференции, посвященной объектно-ориентированному программированию.

Сформулировать его можно в виде простого правила:

тип S - подтип T *тогда и только тогда*, когда

каждому объекту oS типа S соответствует некий объект oT типа T таким образом, что для всех программ P, реализованных в терминах T, поведение P не будет меняться, если oT заменить на oS.

или **S ϵ T < - > Ɐ oS ϵ S ∃ oT ϵ T -> Ɐ P: P<oT> = P<oS>**

Проверка абстракции на тип

Код проверки абстракции на тип на примере нарушения принципа открытости/закрытости.

Класс Repository нарушает и принцип замещения Лисков.

Дело в том, что внутри класса Repository мы оперируем не только абстрактной сущностью AbstractEntity, но и унаследованными типами.

А это значит, что в данном случае подтипы AccountEntity и RoleEntity не могут быть заменены типом, от которого они унаследованы.

По определению имеем нарушение.

Надо заметить, что принципы проектирования взаимосвязаны.

Нарушение одного из принципов скорее всего приведет к нарушению одного или нескольких других принципов.

Наследование

С наследованием не все так просто.

Во-первых, наследование – это одна из самых сильных связей в ОО-мире, которая крепко привязывает наследников к базовому классу

Во-вторых, не всегда легко ответить два вопроса:

1) когда наследование уместно и

2) как его реализовать корректным образом.

Принцип подстановки Лисков призван помочь в корректной реализации наследования, что также должно помочь отказаться от наследования, если его корректная реализация невозможна.

Наследование обычно моделирует отношение "ЯВЛЯЕТСЯ" (IS-A Relationship) между классами.

Говорят, что экземпляр наследника также ЯВЛЯЕТСЯ экземпляром базового класса, что выражается в возможности использования экземпляров наследника везде, где ожидается использование базового класса.

ПРИМЕЧАНИЕ

Бертран Мейер в своей книге "Объектно-ориентированное конструирование программных систем" (глава 24) приводит 12 (!) различных видов наследования, включая наследование реализации (закрытое наследование), IS-A, Can-Do (реализация интерфейсов) и т.п.

Формулировка принципа LSP от Боба Мартина повторяет определение отношения «ЯВЛЯЕТСЯ»:

Принцип подстановки Лисков: *должна существовать возможность использовать объекты производного класса вместо объектов базового класса.*

Это значит, что объекты производного класса должны вести себя согласованно, согласно контракту базового класса.

Какую проблему мы пытаемся решить?

Основной смысл любой иерархии наследования в том, что она позволяет использовать базовые классы, не задумываясь о том, экземпляр какого конкретного класса используется.

Если же реализация (т.е. наследники) не будет знать о протоколе «абстракции» или не будет ему следовать, то в приложении мы будем вынуждены обрабатывать конкретную реализацию специальным образом, что сводит на нет идею использования «абстракции» и наследования.

Почему важно следовать принципу подстановки Лисков?

«Поскольку в противном случае иерархии наследования приведут к неразберихе.

- Неразбериха будет заключаться в том, что передача в метод экземпляра класса-наследника приведет к странному поведению существующего кода.
- Поскольку в противном случае юнит-тесты базового класса никогда не будут проходить для наследников.»

Классический пример: квадраты и прямоугольники

Наследование моделирует отношение «ЯВЛЯЕТСЯ»

В зависимости от «контракта» сущностей (не важно, явного или нет) мы можем говорить о возможности использования наследования.

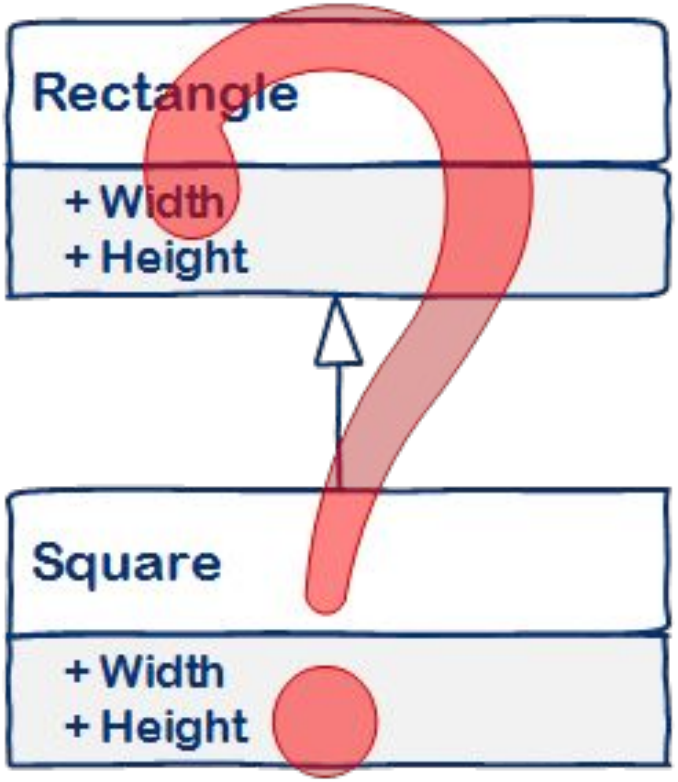
Рассмотрим более подробно известный пример с квадратами и прямоугольниками.

Чтобы понять, будет ли нарушать данная иерархия классов LSP, нужно постараться сформулировать контракты этих классов:

Контракт прямоугольника (инвариант): ширина и высота положительны.

Контракт квадрата (инвариант): ширина и высота положительны; ширина и высота равны.

Квадрат является прямоугольником, но актуально ли это отношение для классов **Rectangle** и **Square**



Код примера

```
class Rectangle {
 int getHeight() const;
 void setHeight(int value);
 int getWidth() const;
 void setWidth(int value); };
class Square extends Rectangle { };
```

Тогда имеем возможность получить при тестировании

```
void invariant(Rectangle  r) {
r.setHeight(200);
r.setWidth(100);
assert(r.getHeight() == 200 and r.getWidth() == 100); }
```

Пока нельзя сказать, являются ли контракты согласованными.

Поскольку у квадрата все стороны равны, то изменение его ширины должно приводить и к изменению его высоты, и наоборот.

Это значит, контракт свойств **Width** и **Height** квадрата становится несогласованным с контрактом этих свойств прямоугольника! (С точки зрения клиента прямоугольника свойства **Width** и **Height** полностью независимы, а значит замена прямоугольника квадратом во время исполнения нарушит это предположение клиента.)

Неизменяемость -immutable

Но это не значит, что данная иерархия наследования является невозможной. **Квадрат перестает быть нормальным прямоугольником, ТОЛЬКО если квадрат и прямоугольник являются изменяемыми!**

Так, если мы сделаем их неизменяемыми), то проблема с контрактами, принципом подстановки и нарушением поведения клиентского кода при замене прямоугольников квадратами пропадет.

Если клиент не может изменить ширину и высоту, то его поведение будет одинаковым как для квадратов, так и для прямоугольников!

Лучше точнее описывать контракт базовых классов

Данный пример показывает несколько важных моментов:

- Во-первых, именно наличие контракта позволяет четко понять, нарушает ли производный класс LSP или нет.
- А во-вторых, пример с неизменяемостью показывает пользу неизменяемости в ОО мире за пределами параллелизма: контракт неизменяемых типов проще, поскольку контролируется лишь конструктором и инвариантом класса.
- В-третьих, этот пример показывает, почему некоторые специалисты рекомендуют, чтобы классы были либо абстрактными, либо запечатанными, без возможности инстанциировать классы из середины иерархии наследования.

Классическим решением проблемы квадратов/прямоугольников является выделение промежуточного абстрактного класса «четырехугольника», от которого уже наследуются квадрат и прямоугольник.

Принцип подстановки Лисков и контракты

Не будет преувеличением сказать, что лишь с помощью принципов Проектирования по контракту вы можете точно понять, что представляет собой наследование. Бертран Мейер «Объектно-ориентированное конструирование программных систем», раздел 16.1.

Существует несколько способов описать ожидаемое поведение типа, т.е. его спецификацию. Для этого мы можем использовать комментарии юнит-тесты или контракты (заставить бы всех ими пользоваться!).

Пример с квадратами и прямоугольниками показал, что мы не можем доказать, нарушает ли конкретный класс принцип подстановки Лисков или нет, пока не определимся с тем, что ожидают клиенты от поведения его базового класса.

LSP – обосновывается контрактами

Если посмотреть на исходное описание принципа подстановки в трудах Барбары Лисков, то можно с удивлением обнаружить, что оно полностью основано таких понятиях, как предусловия, постусловия и инварианты.

Описание этого принципа полностью основано на **принципах проектирования по контракту**:

- Производные классы не должны усиливать предусловия (не должны требовать большего от своих клиентов).
- Производные классы не должны ослаблять постусловия (должны гарантировать, как минимум тоже, что и базовый класс).
- Производные классы не должны нарушать инварианты базового класса (инварианты базового класса и наследников суммируются)
- Производные классы не должны генерировать исключения, не описанные базовым классом.
- Попытка формализации обязанностей класса в терминах предусловий, постусловий и инвариантов позволит более четко сказать, является ли поведение наследника согласованным или нет.

Заключение

Принцип подстановки Лисков не является панацеей в вопросах наследования, он лишь помогает формализовать, в каких пределах может варьироваться поведение наследника с точки зрения контракта базового класса.

В своих трудах Барбара Лисков строила свой анализ на основе контрактов класса: предусловий, постусловий и инвариантов.

И именно с помощью контрактов мы можем хотя бы с некоторой долей уверенности утверждать, что поведение наследника и базового класса является согласованным.

Одиночка

- **Суть паттерна**

Одиночка — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

- **Проблема**

Одиночка решает сразу две проблемы (нарушая *принцип единственной ответственности* класса).

1. Гарантирует наличие единственного экземпляра класса.

Чаще всего это полезно для доступа к какому-то общему ресурсу, например, базе данных.

Представьте, что вы создали объект, а через некоторое время, пробуете создать ещё один. В этом случае, хотелось бы получить старый объект, вместо создания нового.

Такое поведение невозможно реализовать с помощью обычного конструктора, так как конструктор класса **всегда** возвращает новый объект.

2. Предоставляет глобальную точку доступа.

Это не просто глобальная переменная, через которую можно достучаться к определённому объекту.

Глобальные переменные не защищены от записи, поэтому любой код может подменять их значения без вашего ведома.

Но есть и другая грань этой проблемы.

Неплохо бы хранить в одном месте и код, который решает проблему №1, а также иметь к нему простой и доступный интерфейс.

Решение

Все реализации одиночки сводятся к тому, чтобы скрыть конструктор по умолчанию и создать публичный статический метод, который и будет контролировать жизненный цикл объекта-одиночки.

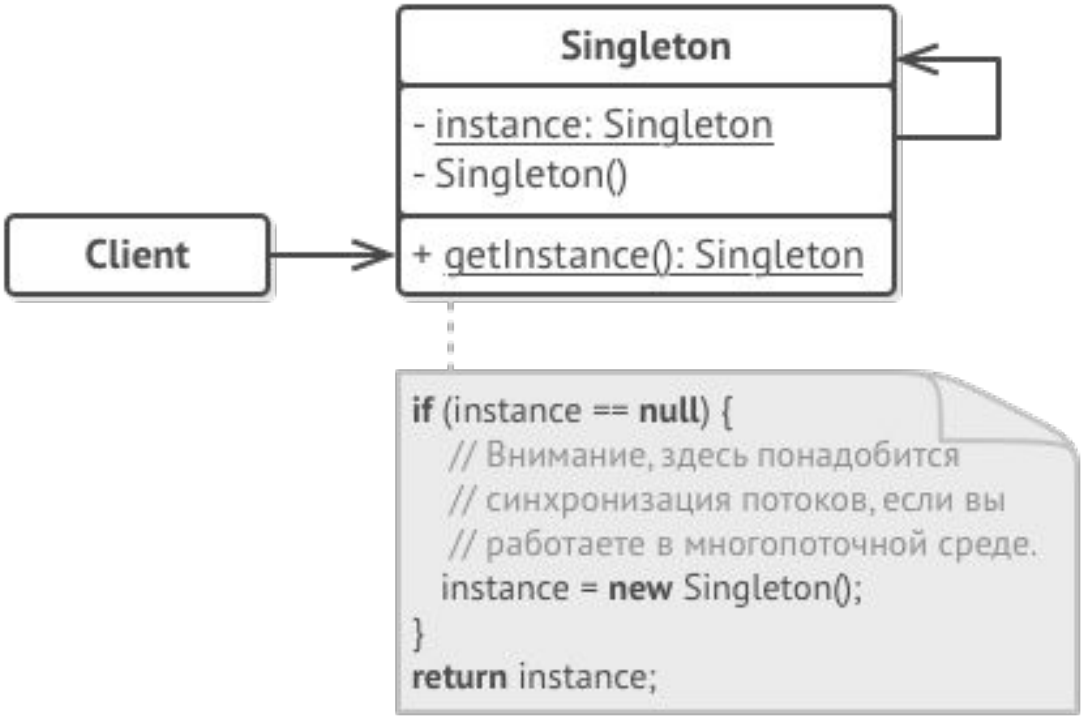
Если у вас есть доступ к классу-одиночке, значит, будет доступ и к этому статическому методу.

Из какой точки кода вы бы его не вызвали, он всегда будет отдавать один и тот же объект.

Структура

одиночка определяет статический метод getInstance(), который возвращает единственный экземпляр класса *Одиночки*.

Конструктор одиночки должен быть скрыт от клиентов. Вызов getInstance() должен быть единственным способом получить объект этого класса.



Пример

- В этом примере роль Одиночки играет класс подключения к базе данных.
- Этот класс не имеет публичного конструктора, поэтому единственный способ получить его объект — это вызвать метод getInstance.
- Этот метод сохранит первый созданный объект и будет возвращать его при всех последующих вызовах.
- Паттерн Одиночка позволяет клиенту не беспокоиться о получении одного-единственного экземпляра объекта класса.
- Он предоставляет глобальную точку доступа к этому экземпляру — статический метод getInstance.

class Database

```
private Object instance:
 Database static method getInstance()
if (this.instance == null) then acquireThreadLock()
// На всякий случай ещё раз проверим не был ли объект создан
 // другим потоком, пока текущий ждал освобождения блокировки
 if (this.instance == null) then this.instance = new Database()
 return this.instance
 private constructor Database()
 // Здесь может жить код инициализации подключения к серверу баз
     данных. // ...
public method query(sql)
// Все запросы к базе данных будут проходить через этот метод.
     Поэтому
// имеет смысл поместить сюда какую-то логику кеширования.
 // ...
```

class Application

```
method main()
 Database foo = Database.getInstance()
foo.query("SELECT ...")
 // ...
Database bar = Database.getInstance()
 bar.query("SELECT ...")
 // В переменной bar содержится тот же
    объект, что и в foo.
```

Шаги реализации

Добавьте в класс приватное статическое поле, которое будет содержать одиночный объект.

Объявите статический создающий метод, который будет использоваться для получения одиночки.

Добавьте «ленивую инициализацию» (создание объекта при первом вызове метода) в создающий метод одиночки.

Сделайте конструктор класса приватным.

В клиентском коде замените вызовы конструктора вызовами создающего метода.

Преимущества и недостатки

(+)

Гарантирует наличие единственного экземпляра класса.

Предоставляет к нему глобальную точку доступа.

Реализует отложенную инициализацию объекта-одиночки.

(-)

Нарушает *принцип единственной ответственности класса*.

Маскирует плохой дизайн.

Проблемы мультипоточности.

Требует постоянного создания Mock-объектов при юнит-тестирования.

Одиночке нашлось применение в стандартных библиотеках Java:

- **java.lang.Runtime#getRuntime()**
- **java.awt.Desktop#getDesktop()**
- **java.lang.System#getSecurityManager()**

Признаки применения паттерна: Одиночку можно определить по статическому создающему методу, который возвращает один и тот же объект.

Singleton.java: Одиночка

```
public final class Singleton {
private static Singleton instance;
 public String value;
 private Singleton(String value) {
// Этот код эмулирует медленную инициализацию.
 try { Thread.currentThread().sleep(1000); }
 catch (InterruptedException ex) { ex.printStackTrace();} this.value =
    value; }
 public static Singleton getInstance(String value) {
if (instance == null) {
instance = new Singleton(value);
 } return instance;
} }
```

DemoSingleThread.java: Клиентский код

```
public class DemoSingleThread {
public static void main(String[] args) {
System.out.println("If you see the same value, then singleton
    was reused (yay!)" + "\n" + "If you see different values, then
    2 singletons were created (booo!!)" + "\n\n" + "RESULT:" +
    "\n");
Singleton singleton = Singleton.getInstance("FOO");
 Singleton anotherSingleton = Singleton.getInstance("BAR");
System.out.println(singleton.value);
System.out.println(anotherSingleton.value);
} }
```

OutputDemoSingleThread.txt: Результаты выполнения

If you see the same value, then singleton was reused (yay!)

If you see different values, then 2 singletons were created (booo!!)

RESULT:

FOO

FOO

Упр. 4.1 Наделите свойством одиночки директора паттерна строитель

Литература

- Бертран Мейер Объектно - ориентированное конструирование программных систем
- https://refactoring.guru/ru/design-patterns/singleton
- Правильный Singleton в Java https://habr.com/ru/post/129494/