CC BY-SA 3.0

Современные средства разработки ПО

Принципы SOLID. Антипаттерны.

МГТУ им. Н.Э. Баумана, ИУ-6, Фетисов Михаил Вячеславович
fetisov.michael@bmstu.ru

Принципы SOLID

- **The Single Responsibility Principle** (SRP) – Принцип единственной ответственности
- **The Open Closed Principle** (OCP) – Принцип открытости/закрытости
- **The Liskov Substitution Principle** (LSP) – Принцип подстановки Барбары Лисков
- **The Interface Segregation Principle** (ISP) – Принцип разделения интерфейса
- **The Dependency Inversion Principle** (DIP) – Принцип инверсии зависимостей

Принцип единственной ответственности (SRP)

- Каждый класс выполняет лишь одну задачу
- ШП Функциональный дизайн
- Антипаттерн «Божественный объект»
- Способы достижимости:
  - использование приёма рефакторинга «выделение класса»;
  - шаблон «фасад»;
  - интерфейсы.
- Примеры:
  - Система управления поливом и клапан (из первого примера ООП);
  - Ведение боя и правила боя (из второго примера ООП).

SRP — пример

```
class Animal {
    constructor(name: string){ }
    getAnimalName() { }
    saveAnimal(a: Animal) { }
}
```

Класс Animal, представленный здесь, описывает какое-то животное. Этот класс нарушает принцип единственной ответственности. Как именно нарушается этот принцип?

SRP — пример 2

- Для того чтобы привести вышеприведённый код в соответствие с принципом единственной ответственности, создадим ещё один класс, единственной задачей которого является работа с хранилищем, в частности — сохранение в нём объектов класса Animal:

```
class Animal {
    constructor(name: string){ }
    getAnimalName() { }
}
class AnimalDB {
    getAnimal(a: Animal) { }
    saveAnimal(a: Animal) { }
}
```

Принцип открытости/закрытости (OCP)

- «Программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения»
- Специфика:
  - близок к SRP;
  - принцип позволяет избежать пересмотра связанного кода, модульных тестов, текстов самодокументирования и других артефактов ПО;
  - добиться снижения трудозатрат.
- Способы достижимости:
  - те же, что для SRP

OCP — пример

- Представим, что у нас есть магазин. Мы даём клиентам скидку в 20%, используя такой класс:

```
class Discount {
    giveDiscount() {
        return this.price * 0.2
    }
}
```

OCP — пример

- Теперь решено разделить клиентов на две группы. Любимым (fav) клиентам даётся скидка в 20%, а VIP-клиентам (vip) — удвоенная скидка, то есть — 40%. Для того, чтобы реализовать эту логику, было решено модифицировать класс следующим образом:

```
class Discount {
    giveDiscount() {
        if(this.customer == 'fav') {
            return this.price * 0.2;
        }
        if(this.customer == 'vip') {
            return this.price * 0.4;
        }
    }
}
```

OCP — пример

- Для того чтобы переработать этот код в соответствии с принципом открытости-закрытости, добавим в проект новый класс, расширяющий класс Discount. В этом новом классе мы и реализуем новый механизм:

```
class VIPDiscount: Discount {
    getDiscount() {
        return super.getDiscount() * 2;
    }
}
```

OCP — пример

- Если решено дать скидку в 80% «супер-VIP» клиентам, выглядеть это должно так:

```
class SuperVIPDiscount: VIPDiscount {
    getDiscount() {
        return super.getDiscount() * 2;
    }
}
```

Принцип подстановки Барбары Лисков (LSP)

- «Объекты в программе должны быть заменяемыми на экземпляры их подтипов без нарушения выполнения программы». Наследующий класс должен дополнять, а не изменять базовый
- Следствия:
  - нельзя усиливать предусловия и ослаблять постусловия методов производных классов;
  - нельзя создавать новых мутаторов свойств, не предусмотренных базовым классом;
  - производные классы не должны бросать исключения, не предусмотренные в базовом классе.
- Примеры:
  - ?

LSP

- В последующей статье Лисков кратко сформулировала свой принцип следующим образом:
  - Пусть **q(x)** является свойством, верным относительно объектов **x** некоторого типа **T**. Тогда **q(y)** также должно быть верным для объектов **y** типа **S**, где **S** является подтипом типа **T**.
- Роберт С. Мартин определил этот принцип так:
  - Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.

Принцип разделения интерфейса (ISP)

- «Много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения»
  - или «Программные сущности не должны зависеть от методов, которые они не используют»
- Особенности:
  - улучшает адаптивность кода;
  - существенно упрощает рефакторинг.
- Примеры:
  - те же, что для SRP

Принцип инверсии зависимостей (DIP)

- Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
- Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.
- Примеры:
  - те же, что для SRP;
  - задачи про парник и бой на первых семинарах.

Антипаттерны

- **Паттерн проектирования** (шаблон проектирования)
- **Спагетти-код**
  - Дейкстра, Эдсгер Вибе (Edsger Wybe Dijkstra. "Go To Statement Considered Harmful". Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148.)
- **Божественный объект** (блоб/blob) — Концентрация слишком большого количества функций в одной части системы (классе).
- **Базовый класс-утилита** — Наследование функциональности из класса-утилиты вместо делегирования к нему.
- **Вызов предка** — Для реализации прикладной функциональности методу класса-потомка требуется в обязательном порядке вызывать те же методы класса-предка.
- **Одиночество** — Неуместное использование паттерна одиночка.
- **Френд-зона** — Неуместное использование дружественных классов и дружественных функций в языке C++.
- **Каша из интерфейсов** — Объединение нескольких интерфейсов, разделенных согласно принципу изоляции интерфейсов (ISP), в один.

Антипаттерны (продолжение)

- **Лазанья-код** — Чрезмерное связывание между собой уровней абстракции, приводящее к невозможности изменения одного уровня без изменения остальных.
- **Равиоли-код** — Объекты настолько «склеены» между собой, что практически не допускают рефакторинга.
- **Копи-паст код** — Копирование (и лёгкая модификация) существующего кода вместо создания общих решений.
- **Золотой молоток** — Сильная уверенность в том, что любимое решение универсально применимо.
- **Фактор невероятности** — Предположение о невозможности того, что сработает известная ошибка.
- **Дым и зеркала** — Демонстрация того, как будут выглядеть ненаписанные функции.

Вопросы?