Принципы SOLID

- **S** — Принцип единой ответственности (SRP)
- **O** — Принцип открытости/закрытости (OCP)
- **L** — Принцип заменяемости (LCP)
- **I** — Принцип разделения интерфейсов (ISP)
- **D** — Принцип инверсии зависимостей (DIP)

Анализ и проектирование

Анализ – это **исследование** задачи и требований к её реализации. Ключевой момент, это именно исследование, мы исследуем задачу, то, чего, например от нас хочет заказчик.

Проектирование – это **концептуальное решение** задачи, удовлетворяющее всем требованиям к ней.

https://www.youtube.com/watch?v=ph6IleGTmmw

Объектная декомпозиция

Основная задача проектирования при объектной декомпозиции – это выполнение двух условий.

Условия декомпозиции:

- **Высокое сцепление** (high cohesion): максимизация связей внутри классов
- **Низкая связанность** (low coupling): минимизация связей между классами

Два главных принципа ООП

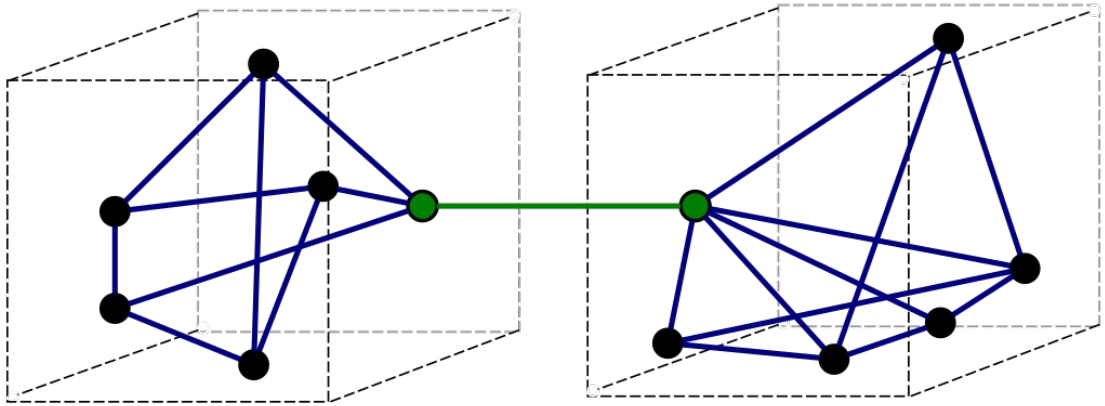
«Минимизация связей между классами» не означает «отказ от связей вообще» - в этом нет смысла, связи так или иначе будут.

«Минимизация связей между классами» означает не отказ от них, а их ослабление.

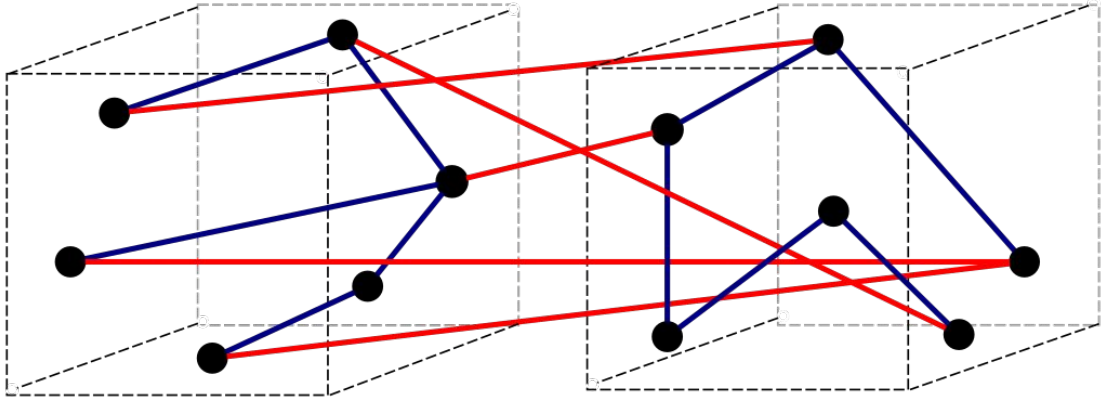
Сильное сцепление и слабая связанность – это два главных принципа ООП.

https://medium.com/german-gorelkin/low-coupling-high-cohesion-d36369fb1be9

Low coupling, high cohesion!



a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

Понятие архитектуры

Архитектура – это понимание того, как система разделена на компоненты, и как эти компоненты взаимодействуют друг с другом посредством интерфейсов.

Также **архитектура** - это набор решений проектирования, изменить которые считается сложным.

То есть архитектура – это нечто необратимое для проекта в целом. Отсюда её важность и столько разговоров о ней.

Гибкость системы

Чем меньше в нашей системе необратимого, тем лучше. Парадоксально, но получается, что чем меньше у нас будет архитектуры (то есть необратимых решений), тем лучше.

Чем больше в нашей системе гибкости, тем она лучше написана.

Качество архитектуры

Как определить качество архитектурного решения?

Нужно задать себе вопрос: «А что, если я ошибся, и мне придется изменить это решение в будущем, - какие будут последствия?» Вопрос не о самом качестве, а о том, архитектурное оно вообще или нет?

Если некоторое решение используется по всему приложению, то стоимость его изменения будет огромной, и значит решение является архитектурным. Роберт Мартин говорил: «Хорошая архитектура позволяет откладывать принятие ключевых решений».

Чем больше в системе гибких (т.е. обратимых, динамически задаваемых) решений - тем она лучше.

Рекомендации к архитектуре

- Подумать, что придется реализовать жёстко и явно, а что можно реализовать динамически
- Всё, что придется реализовать жёстко и явно (то есть архитектуру), по крайней мере сделать хорошо, помня о цене вопроса
- Всё, что можно реализовать динамически, именно так и реализовать. Это наша цель.

  А «динамически» — это же полиморфизм! Это возможность отложить наше решение на потом. Сейчас не знаем точно тип объектов, которые нам понадобятся. Вместо этого используем абстракцию, а конкретику подставим потом. Это и есть способ отложить решение на потом.

Итого

- Анализ – это исследование
- Проектирование – это концептуальное решение
- Архитектура – это понимание разделения системы на компоненты
- Главное во всем этом – сильное внутреннее сцепление и слабая внешняя связанность

Абстрагирование

Абстрагирование – это выделение существенных характеристик объекта и отбрасывание несущественных. Абстракция – это обобщение. А обобщать мы можем в любой момент времени. Теряем в детализации, зато выигрываем в охвате.

Абстр.классы и интерфейсы

Абстрактный класс – это **заготовка** для будущих классов. Мы делегируем реализацию тех методов, которые не определены, производным классам.

Интерфейс – это **контракт** для будущих классов. Это набор обязательств для каких-то классов, которые если берут эти обязательства, то должны их реализовать. Интерфейс это не просто контракт, это всегда **публичный** контракт. Интерфейс не наследуют, а **реализуют**. При этом класс может выполнять несколько разных контрактов, а значит может реализовывать несколько разных интерфейсов. Главное – это работа с объектами через интерфейсные указатели. Интерфейс – это максимально абстрактная сущность в ООП. Чистейший, незамутнённый динамический полиморфизм. Интерфейсы не являются заменой множественного наследования в тех языках, где оно не реализовано.

Пример абстрактного класса

https://bit.ly/3K9fWeE

https://gist.github.com/sunmeat/777658c60dbebc4cffd716050ff84f94

Пример интерфейса

https://bit.ly/3nonZdO

https://gist.github.com/sunmeat/4752da83f5198bd762158ef538c19a55

Реализация 2 интерфейсов

```
struct IA
{
    virtual void fa() = 0;
};

struct IB
{
    virtual void fb() = 0;
};

class MyClass : public IA, public IB
{
public:
    virtual void fa() { ... }   // Реализация IA
    virtual void fb() { ... }   // Реализация IB
};
```

ООП в 3 строчках :)

```
MyClass obj;
IA *pa = &obj;
pa->fa();
```

```
IA *pa = new MyClass;
pa->fa();
```

Разница между Интерф. и АК

С одной стороны, ничего общего: один – набор требований, чистая абстракция, а второй – по сути обычный, но немного недоделанный класс.

Но с другой стороны, с точки зрения ООП-моделирования – наоборот, между интерфейсами и абстрактными классами никакой особой разницы нет.

И то, и другое определяет «абстрактный интерфейс» для семейства объектов, и разница лишь в том, есть ли поведение по умолчанию.

Итого

- Абстрактный класс – это заготовка
- Интерфейс – это контракт
- Интерфейсы дают нам максимальную возможность абстрагироваться от типа объекта во время работы программы

Принципы S.O.L.I.D.

Что такое S.O.L.I.D?

- Это принципы проектирования классов
- Всего их 5 штук

Основные идеи

1. Увеличение уровня абстракции: интерфейсы, а не реализации
2. Усиление связей внутри классов
3. Ослабление связей между классами
4. **Хорошо спроектированный класс не потребуется менять в будущем!**

https://www.baeldung.com/solid-principles

SRP

Single Responsibility Principle (SRP) – принцип единственной обязанности, принцип разделения обязанностей (на каждый класс должна быть возложена одна единственная обязанность, а также класс должен иметь одну и только одну причину для изменений).

Именно этот принцип является реализацией условия «сильное внутреннее сцепление». Он как раз и занимается усилением связей внутри класса.

SRP: bad example

https://bit.ly/3K7al8x

https://gist.github.com/sunmeat/3caae0b8fa625ded1d8d92f3bc9ceb9e

SRP: good example

https://bit.ly/3Fq5Rq6

https://gist.github.com/sunmeat/599c1e9e6d55302af4aaf4b2b493f308

SRP

На предыдущем слайде показан простой пример и тут очевидно, сколько и какие обязанности были у исходного класса Order. В реальной ситуации выделение у класса нескольких ответственностей может быть гораздо менее очевидным и однозначным.

Для таких случаев существует правило: старайтесь выделять ответственности, которые изменяются независимо друг от друга (тогда у каждого будет та самая одна причина для изменений).

SRP

В примере выше, если вдруг изменится способ вывода заказа на печать, то достаточно будет поправить только класс PrintManager, а два других изменять не нужно будет.

Сложность в применении данного принципа заключается в том, что прежде всего нужно научиться правильно чувствовать границы его использования.

То есть, научиться не доводить ситуацию до абсурда: не нужно перебарщивать с выделением каждого чиха в отдельный класс. Поэтому тут же говорят и об анти-паттерне “принцип размытой ответственности”.

OCP

Open/Closed Principle (OCP) – принцип открытости/закрытости (программные сущности должны быть открыты для расширения, но закрыты для изменения.

Проблема: надо добавить функциональность – приходится менять класс. А это противоречит идее S.O.L.I.D.

Решение: абстракция и интерфейсные указатели.

Современное объектно-ориентированное проектирование, не отрицая важность наследования, делает ставку на композицию, и на передачу интерфейсных указателей, поэтому, когда мы говорим про расширение, мы говорим про интерфейсные указатели.
Принцип открытости/закрытости требует переходить от реализации к абстракции.

OCP: bad example

https://bit.ly/3Ftsr19

https://gist.github.com/sunmeat/93b6d02fcfe13953f47c9af3598a574b

OCP: good example

https://bit.ly/33yxtw7

https://gist.github.com/sunmeat/2478549dbb21550ef4e35671bc5c6b9f

OCP

Суть принципа OCP в том, что единожды созданные классы не следует изменять под конкретные нужды конкретной ситуации.

Для изменения поведения некоторого класса необходимо явным образом описать его интерфейс, и создать другую реализацию этого интерфейса.

Перебарщивание с этим принципом также приводит к анти паттерну: принципу фабрики фабрик.

LSP

Liskov Substitution Principle (LSP) – принцип замещения (функции, которые используют указатели на базовые классы, **должны** иметь возможность использовать объекты производных классов, не зная об их конкретных типах.
Для создания взаимозаменяемых частей эти части должны соответствовать контракту, который позволяет заменять эти части друг другом. Этот принцип на самом деле говорит про наследование, о том, как правильно наследоваться, и при этом как правильно абстрагироваться.

Проблемы:

1.Проверка абстракции на конкретный тип

2.Ошибочное наследование. Нарушение контракта при реализации интерфейса (смотрите видео Димы, начиная с 1:38:00).

В результате мы должны вставлять костыль и обратно уменьшать уровень абстракции.

LSP: bad example

https://bit.ly/3rguJLS

https://gist.github.com/sunmeat/07b01f99bba5b2150a3fc7f53967d6be

LSP: good example

**Достаточно перенести вызов метода LoadExcelLibrary()
в тело метода GenerateReport() класса ExcelReporter.**

LSP

То есть, нужно корректно реализовывать публичный контракт интерфейса **не только физически** (переопределяя абстрактные методы), **но и логически** (например, сколько раз вызвали Add, столько и должен возвращать GetCount).

То есть реализация интерфейса – это не только определение его методов, но и следование при этом предполагаемой им логике. Принцип замещения Б.Лисков, запрещает опускаться обратно от абстракции к реализации. Здесь антипаттерн – это Anti-LSP: Принцип непонятного наследования. Проявляется либо в чрезмерном количестве наследования, либо в его полном отсутствии, в зависимости от опыта и взглядов местного главного архитектора.

ISP

Interface Segregation Principle (ISP) – принцип разделения интерфейса – клиент, реализующий интерфейс, не должен вынуждено зависеть от элементов интерфейса, которые он не использует.

Много специализированных интерфейсов лучше, чем один универсальный.

Проблема: не все методы, которые описывает интерфейс нужны в тех классах, которые его имплементируют, поэтому нужно разделить интерфейс на более мелкие, и имплементировать только те, которые необходимы.

ISP: bad example

https://bit.ly/3rp6pYu

https://gist.github.com/sunmeat/bc829ddad45261695fcb9a1c189921cb

ISP: good example

https://bit.ly/3qqm9Le

https://gist.github.com/sunmeat/6eacb206150ced7ea786cbf7ba792c67

ISP

Мы обязываем класс интерфейс реализовывать, а он не нужен весь. Если ваш интерфейс охватывает больше возможностей, чем вам реально нужно, в большинстве случаев это говорит о том, что следует разбить его на несколько.

Анти-паттерн здесь — это Anti-ISP: принцип тысячи интерфейсов. Это ситуация, когда на каждый чих создают слишком много маленьких и абсурдно частных интерфейсов, что делает их неудобными для использования всеми клиентами.

DIP

Dependency Inversion Principle (DIP) – принцип инверсии зависимостей, который говорит, что:

- 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.

Признаки слишком высокой зависимости архитектуры:

- **Жёсткость**. Меняем в одном месте – вынуждены менять в куче других мест.
- **Хрупкость**. Меняем в одном месте – начинаются ошибки в куче других мест.
- **Неподвижность**. Сложно вынести код в другой проект.

Причина этих проблем в том, что между классами возникает слишком много жёстко прописанных зависимостей.

DIP

Инверсия зависимостей предлагает избавиться от жёстко прописанных зависимостей между классами, инвертировав их в зависимости от абстракций. Суть здесь в инверсии сильных зависимостей от реализаций в слабые зависимости от абстракций.

Чуть упрощённая, но действенная интерпретация принципа DIP выражается эвристическим правилом «зависеть надо от абстракции». Это значит, что не должно быть зависимостей от конкретных классов. Вместо этого типы указателей на зависимости должны быть интерфейсами или абстрактными классами. Главным плюсом инвертирования зависимостей является инвертирование владения: не мы владеем объектом – не мы за него отвечаем. Мы на этот объект просто указываем. А значит, зависимость от него (связь с ним) – слабая, как того и требует условие «низкая связанность».

DIP

Паттерны реализации принципа инверсии зависимости, в данном случае паттерны проектирования:

- Factory Method
- Service Locator
- **Dependency Injection**

Паттерн Dependency Injection (внедрение зависимости) – не просто инвертирует сильные зависимости от реализаций в слабые зависимости от абстракций, но и выносит их из класса, предлагая внедрять их в класс извне. Классический вариант внедрения зависимости в наш класс – через конструктор. В результате, мы заменяем композицию агрегацией и перекладываем часть проблем с текущего класса куда-то выше. Именно паттерн Dependency Injection является реализацией условия «Слабая внешняя связанность». Он как раз и занимается ослаблением связей между классами. Программируйте на основе абстракций (интерфейс, абстрактный класс и т.п.), а не реализаций.

DIP: bad example

https://bit.ly/3qrzxir

https://gist.github.com/sunmeat/0a799f98b3bba0aa5eb99bff681c142b

DIP: good example

https://bit.ly/34Sb6Cr

https://gist.github.com/sunmeat/6d1223ca4efca9719a875c89e65b2103

DIP

Анти-паттерн здесь – это Anti-DIP: Принцип инверсии сознания или DI головного мозга. Это ситуация, когда интерфейсы создаются для каждого класса и пачками передаются через конструкторы. Понять, где находится логика, становится практически невозможно. Главное: **программируйте на основе абстракций, а не реализаций, и получайте интерфейсные указатели на готовые зависимости, а не создавайте/уничтожайте их сами.**

В действительности 5-й принцип является прямым следствием 2-го и 3-го. И вообще его можно было отдельно не указывать. Но Роберт Мартин посчитал настолько важным с акцентировать внимание на этом, что вынес его в отдельный принцип.

Пример

```
class XMLHttpService extends XMLHttpRequestService {}

class Http {
    constructor(private xmlhttpService: XMLHttpService) { }
    get(url: string , options: any) {
        this.xmlhttpService.request(url,'GET');
    }
    post() {
        this.xmlhttpService.request(url,'POST');
    }
    //...
}
```

Проблема

Здесь класс Http представляет собой высокоуровневый компонент, а XMLHttpService — низкоуровневый. Такая архитектура нарушает пункт 1 принципа инверсии зависимостей: «Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций».

Класс Http вынужденно зависит от класса XMLHttpService. Если мы решим изменить механизм, используемый классом Http для взаимодействия с сетью — скажем, это будет Node.js-сервис или, например, сервис-заглушка, применяемый для целей тестирования, нам придётся отредактировать все экземпляры класса Http, изменив соответствующий код. Это нарушает принцип открытости-закрытости.

Решение

Создадим абстракцию для низкоуровнего модуля XMLHttpService и теперь модуль Http будет зависить от неё.

```
interface Connection {
    request(url: string, opts:any);
}
```

```
class Http {
    constructor(private httpConnection: Connection) { }
    get(url: string , options: any) {
        this.httpConnection.request(url,'GET');
    }
    post() {
        this.httpConnection.request(url,'POST');
    }
    //...
}
```

Решение

Теперь, вне зависимости от того, что именно используется для организации взаимодействия с сетью, класс Http может пользоваться тем, что ему передали, не заботясь о том, что скрывается за интерфейсом Connection.

```
class XMLHttpService implements Connection {
    const xhr = new XMLHttpRequest();
    //...
    request(url: string, opts:any) {
        xhr.open();
        xhr.send();
    }
}
```

Решение

```
class NodeHttpService implements Connection {
    request(url: string, opts:any) {
        //...
    }
}

class MockHttpService implements Connection {
    request(url: string, opts:any) {
        //...
    }
}
```

Решение

Как можно заметить, здесь высокоуровневые и низкоуровневые модули зависят от абстракций. Класс Http (высокоуровневый модуль) зависит от интерфейса Connection (абстракция). Классы XMLHttpService, NodeHttpService и MockHttpService (низкоуровневые модули) также зависят от интерфейса Connection.

Кроме того, стоит отметить, что следуя принципу инверсии зависимостей, мы соблюдаем и принцип подстановки Барбары Лисков. А именно, оказывается, что типы XMLHttpService, NodeHttpService и MockHttpService могут служить заменой базовому типу Connection.

Краткое описание принципов

1. Single Responsibility Principle – делай модули меньше.
2. Open/Closed Principle – делай модули расширяемыми.
3. Liskov Substitution Principle – наследуйся правильно.
4. Interface Segregation Principle – дроби интерфейсы.
5. Dependency Inversion Principle – используй интерфейсы.

Что каждый принцип делает

- 1-й принцип (SRP) – реализация концепции «high cohesion», 4-й сюда же.
- 2-й (OCP) и 3-й (LSP) принципы и их следствие 5-й принцип (DIP) – реализация концепции «low coupling»
- Абстрагируемся где только можно.

Общие итоги

- Архитектура – это то, что потом СЛОЖНО изменить
- Программируйте интерфейсами, а не реализациями. Абстракция гибче детализации
- S.O.L.I.D. – лучший друг программиста!

Домашнее задание

- Посмотреть видео Дмитрия Барабаша https://www.youtube.com/watch?v=ph6IleGTmmw
- Посмотреть любое другое видео про SOLID
- На майстат прислать скриншот второго видео + возникшие вопросы по теме