

Лучшие практики разработки



ПРЕПОДАВАТЕЛЬ



Юрий Костяной

Java/Kotlin backend-разработчик

- 3+ года опыта в коммерческой разработке
- 2+ года опыта в преподавании
- Проекты по интеграции сторонних платформ, CRM
- Проблемно-ориентированный подход в преподавании



ВАЖНО:

- Камера должна быть включена на протяжении всего занятия.
- Если у Вас возник вопрос в процессе занятия, пожалуйста, поднимите руку и дождитесь, пока преподаватель закончит мысль и спросит Вас, также можно задать вопрос в чате или когда преподаватель скажет, что начался блок вопросов.
- Организационные вопросы по обучению решаются с кураторами, а не на тематических занятиях.
- Вести себя уважительно и этично по отношению к остальным участникам занятия.
- Во время занятия будут интерактивные задания, будьте готовы включить камеру или демонстрацию экрана по просьбе преподавателя.

ПЛАН ЗАНЯТИЯ

1. Основной блок
2. Вопросы по основному блоку
3. Домашняя работа



TEL-RAN
by Starta Institute

1

ПОВТОРЕНИЕ ИЗУЧЕННОГО

Повторение

1 Для чего нужны unit-тесты? В чём их преимущество?

2 Что нужно тестировать в написанном классе?

Ответы и другие вопросы: <https://timmson.github.io/java-interview/016-test.html>



Повторение

1 Для чего нужны unit-тесты? В чём их преимущество?

2 Что нужно тестировать в написанном классе?

1 Для проверки функционирования нового/доработанного элементарного модуля, а также проверки, что выполненные изменения не ломают существующий функционал приложения. Автоматизированное выполнение, скорость выполнения, простота.

2 Все публичные методы, в т.ч. все ветки кода в методах, а также все бросаемые исключения.

Ответы и другие вопросы: <https://timmson.github.io/java-interview/016-test.html>

Повторение

Что будет в результате выполнения
следующего теста?

```
@SuppressWarnings("unused")
@DisplayName("Проверка работы JUnit")
@Disabled
@Test
void helloJUnit5() {
    assertEquals(10, 5 + 5);
}
```


Повторение

Что будет в результате выполнения
следующего теста?

```
@SuppressWarnings("unused")
@DisplayName("Проверка работы JUnit")
@Disabled
@Test
void helloJUnit5() {
    assertEquals(10, 5 + 5);
}
```

Сообщение, что тест проигнорирован, т.к. указана аннотация @Disabled.

StringHandlerTest.helloJUnit5() is @Disabled

Повторение

Исправьте ошибку в тесте так,
чтобы тест проходил.

```
@Test  
void checkCollection() {  
    List<String> strs = List.of("one", "two", "three");  
    assertEquals(strs.isEmpty());  
}
```

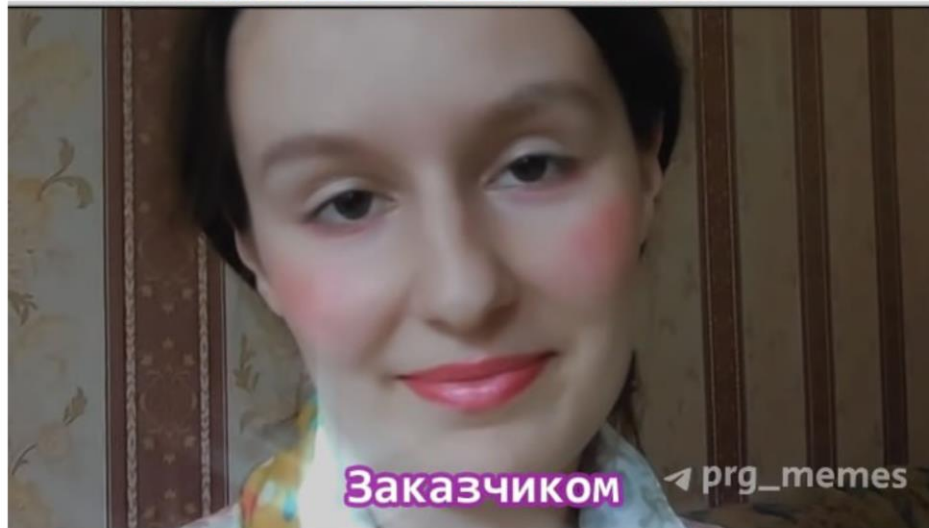
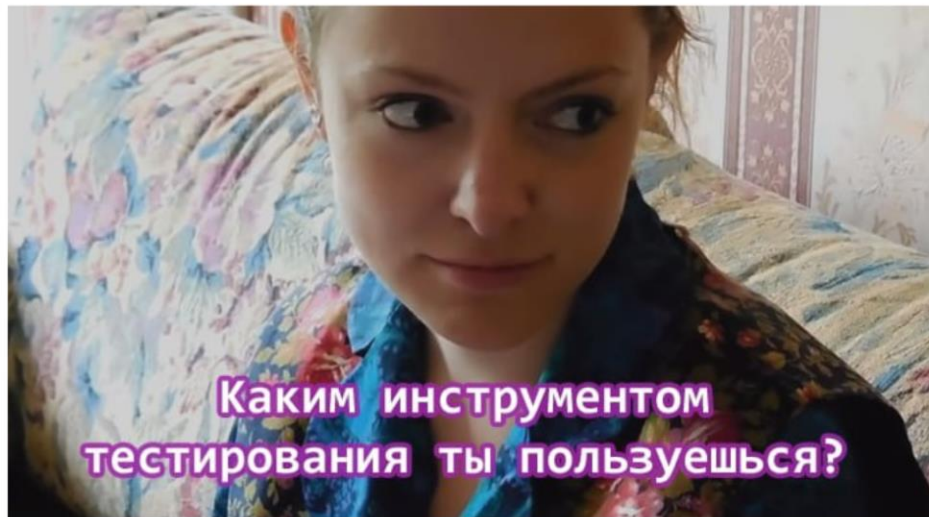
Повторение

Исправьте ошибку в тесте так,
чтобы тест проходил.

```
@Test  
void checkCollection() {  
    List<String> strs = List.of("one", "two", "three");  
    assertFalse(strs.isEmpty());  
}
```

Повторение

В чём прикол мема?



2

ОСНОВНОЙ БЛОК

Введение

- Как завещали предки
- Солидный, крепкий, надёжный
- И всё?
- По шаблонам



Проблема (первая практика)

Программирование в целом и, в частности, программирование на Java существует много лет. На текущий момент сформировано множество полезных принципов и практик, которыми глупо было бы не пользоваться.

изобретáть (изобрестí) велосипéd

изобретать то, что уже изобретено,
решать задачу, которая уже кем-то решена и т.п.



Не надо изобретать велосипед!
Лучше почитай, что об этом пишут
умные люди.

Как завещали предки

Принцип DRY

DRY – don't repeat yourself – принцип, который говорит о том, что в коде не должно быть дублирований. Если при написании кода Вы заметили, что пишете то, что уже писали ранее, то это повод для того, чтобы:

- Вынести повторяющийся код в общий метод.
- Убрать дублирование похожих методов с помощью параметризации.
- Вынести метод/набор методов, дублирующийся в разных классах, в отдельный утилитарный класс.
- Использовать генерацию кода (Lombok, JAXB и т.д.) или сторонние библиотеки.

Вариации: DRY or DIE (duplication is evil), DRY not WET (don't write everything twice). В многопоточности иногда нарушают принцип DRY и переходят к WET, чтобы повысить стабильность работы приложения.



Как завещали предки

Принцип KISS

KISS – Keep It Simple, Stupid – не усложняй! – т.е. стоит делать максимально простой и понятный код. Чем меньше и проще класс, чем проще логика его методов, чем проще архитектура всего приложения, тем проще будет поддерживать код (находить ошибки, добавлять новый функционал). Это значит, что скорость и эффективность работы программистов будет выше. К тому же новым сотрудникам будет проще погружаться в проект.

Вариации:

*«Всё следует упрощать до тех пор, пока это возможно,
но не более того»*

(Альберт Эйнштейн)



Как завещали предки

Принцип YAGNI

YAGNI – You Ain't Gonna Need It – вам это не понадобится!

Суть в том, чтобы реализовать только поставленные задачи и отказаться от избыточного функционала.

Вариации:

Бритва Оккама (иногда лезвие Оккама) – методологический принцип, в кратком виде гласящий: «Не следует привлекать новые сущности без крайней на то необходимости».



YAGNI

It may look like overkill, but I'm sure we'll need it eventually.

Как завещали предки

Принцип POLA

POLA – principle of least astonishment – принцип наименьшего удивления: *если назначение блока кода неясно, то его поведение должно быть наиболее ожидаемым для других программистов.*

Принцип пришёл из эргономики. Изначально создан, чтобы разрабатывать легко поддерживаемые системы.

В каком-то смысле в этом случае код выглядит "скучным" и простым. Вы его читаете, делаете предположения относительно того, что и для чего в нем. И ваши предположения почти всегда оправдываются.



Как завещали предки

Инкапсулируйте то, что меняется

Определите аспекты программы, класса или метода, которые меняются чаще всего, и отделите их от того, что остаётся постоянным.

Этот принцип преследует единственную цель – уменьшить последствия, вызываемые изменениями.

Приложение

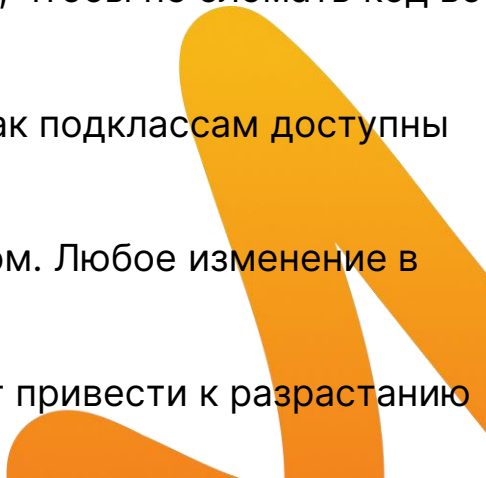
Изменения



Как завещали предки

Предпочитайте композицию наследованию

Наследование – это самый простой и быстрый способ повторного использования кода между классами, но он обладает рядом минусов:

- подкласс не может отказаться от интерфейса или реализации своего родителя;
 - переопределяя методы родителя, нужно заботиться о том, чтобы не сломать код во всех местах использования суперкласса.
 - наследование нарушает инкапсуляцию суперкласса, так как подклассам доступны детали родителя.
 - подклассы слишком тесно связаны с родительским классом. Любое изменение в родителе может сломать поведение в подклассах.
 - повторное использование кода через наследование может привести к разрастанию иерархии классов.
- 

Как завещали предки

Программируйте на уровне интерфейса

Программируйте на уровне интерфейса, а не на уровне реализации.

Код должен зависеть от абстракций (абстрактных классов и интерфейсов), а не конкретных классов.

Поэтому использовать `List<String>` лучше, чем `ArrayList<String>`



Как завещали предки

Программируйте на уровне интерфейса

Когда вам нужно наладить взаимодействие между двумя объектами разных классов, вы можете начать с того, что попросту сделаете один класс зависимым от другого. Но есть и другой, *более гибкий*, способ:

1. Определите, что именно нужно одному объекту от другого, какие методы он вызывает.
2. Затем опишите эти методы в отдельном интерфейсе.
3. Сделайте так, чтобы класс-зависимость имплементировал этот интерфейс. Скорее всего, нужно будет только добавить этот интерфейс в описание класса.
4. Теперь вы можете и сделать второй класс зависимым от интерфейса, а не конкретного класса.

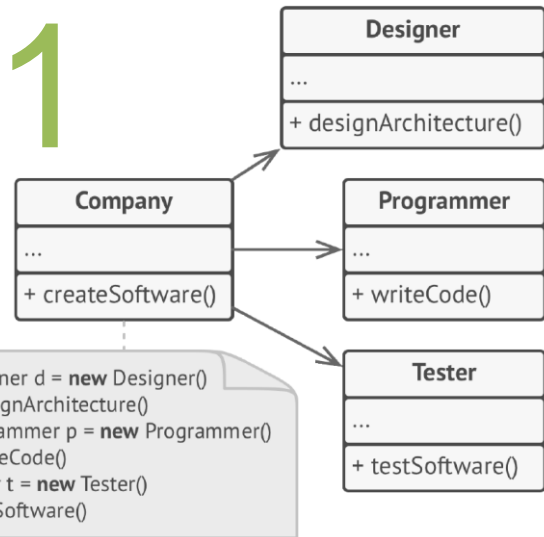
Теперь необходимость заменить реализацию второго класса не вызовет проблем.

Подход широко применяется в паттернах проектирования. Например, в паттерне MVC.

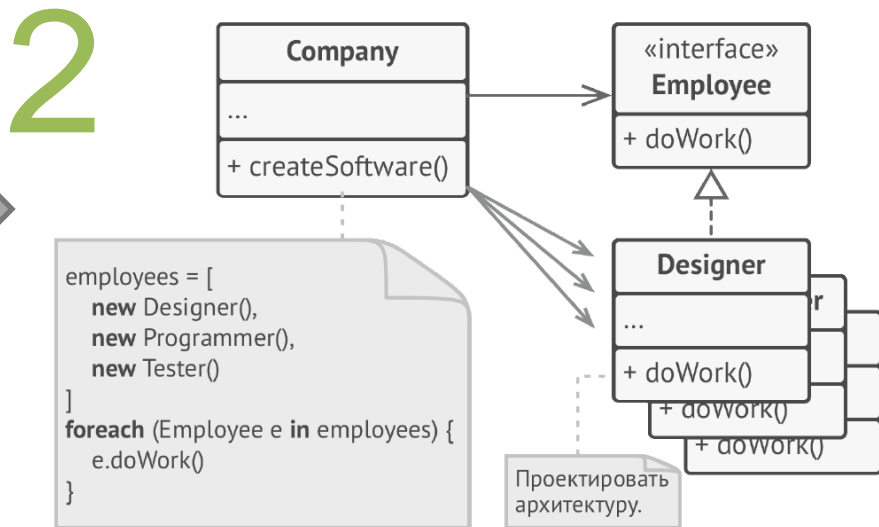
Как завещали предки

Программируйте на уровне интерфейса

Жёстко связанные классы



Зависимые методы вынесены отдельный интерфейс



Класс компании всё ещё остаётся жёстко привязанным к конкретным классам работников. Это не очень хорошо, особенно, если предположить, что нам понадобится реализовать несколько видов компаний. Все эти компании будут отличаться тем, какие конкретно работники в них нужны.

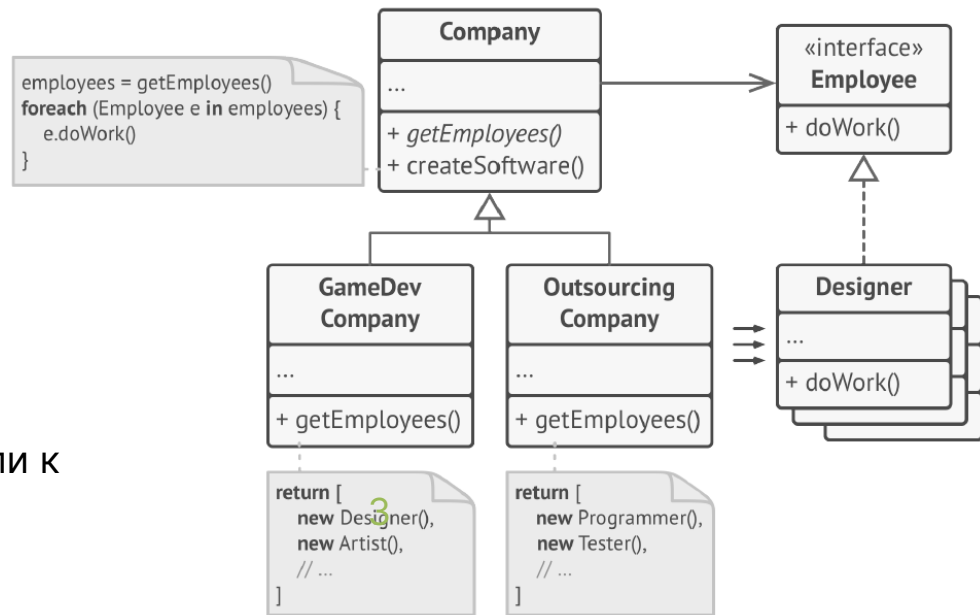
Как завещали предки

Программируйте на уровне интерфейса

Мы можем сделать метод получения сотрудников в базовом классе компании абстрактным. Конкретные компании должны будут сами позаботиться о создании объектов сотрудников. А значит, каждый тип компаний сможет иметь собственный набор сотрудников.

Основной код класса компании стал независимым от классов сотрудников. Конкретных сотрудников создают конкретные классы компаний.

Выполненные преобразования кода привели к реализации паттерна «Фабричный метод»



Солидный, крепкий, надёжный Принципы **SOLID**

Принцип **SOLID** в упрощенном варианте означает, что когда при написании кода используется несколько принципов вместе, то это значительно облегчает дальнейшую поддержку и развитие программы.



Солидный, крепкий, надёжный **Single responsibility principle**

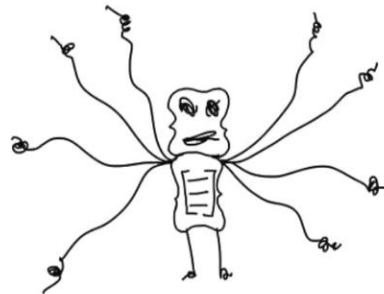
Принцип единственной ответственности – на каждый класс должна быть возложена одна-единственная обязанность (хранить данные, выполнять конвертацию, делать итерацию по коллекции и т.д.).



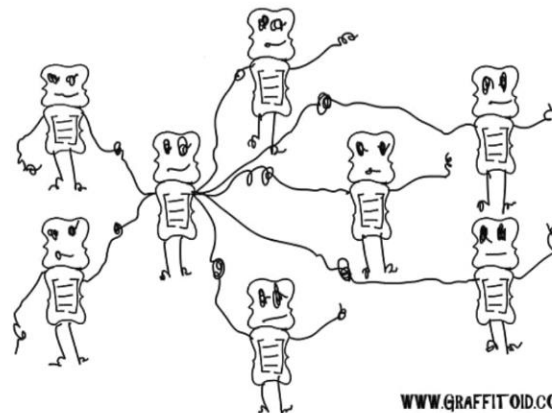
SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

MULTIPLE RESPONSIBILITY

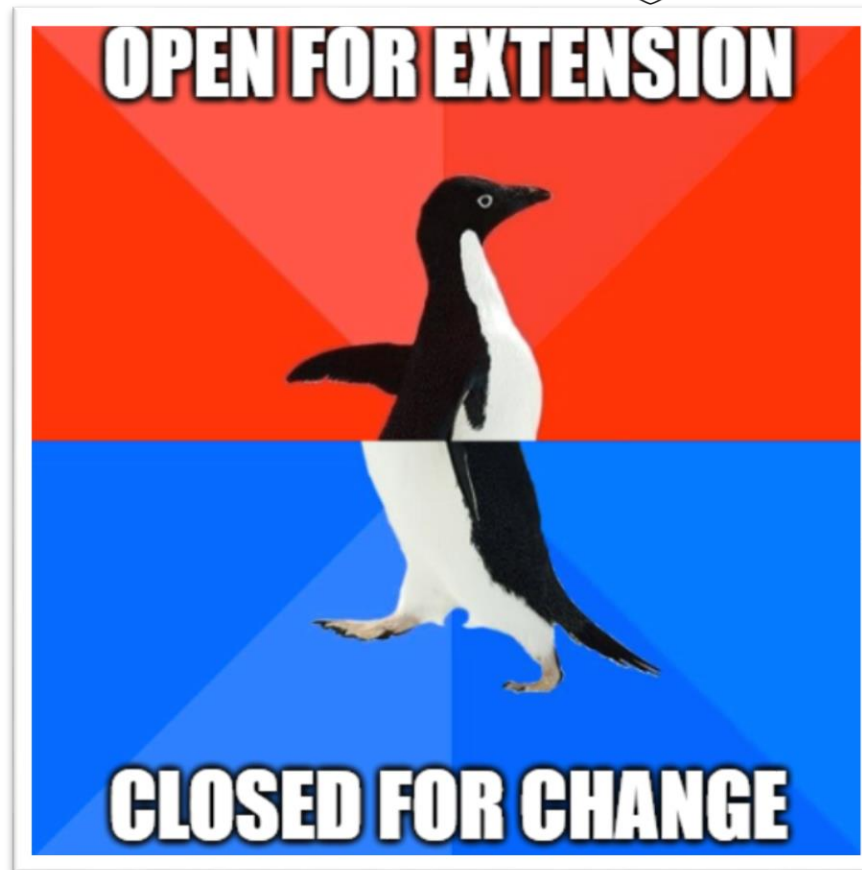


SINGLE RESPONSIBILITY



Солидный, крепкий, надёжный **Open/closed principle**

Принцип открытости/закрытости –
программные сущности должны быть закрыты для изменения, но открыты для расширения. Если класс уже был написан, одобрен, протестирован, возможно, внесён в библиотеку и включён в проект, после этого пытаться модифицировать его содержимое нежелательно. Вместо этого вы можете создать подкласс и расширить в нём базовое поведение, не изменяя код родительского класса напрямую.



Солидный, крепкий, надёжный

Liskov substitution principle



Принцип подстановки Барбары Лисков – подклассы должны дополнять, а не замещать поведение базового класса.

- 1 Типы параметров метода подкласса должны совпадать или быть более абстрактными, чем типы параметров базового метода.
- 2 Тип возвращаемого значения метода подкласса должен совпадать или быть подтипом возвращаемого значения базового метода.
- 3 Метод не должен выбрасывать исключения, которые не свойственны базовому методу.
- 4 Метод не должен ужесточать `_пред_условия`.
- 5 Метод не должен ослаблять `_пост_условия`.
- 6 Инварианты класса должны остаться без изменений. Инвариант – это набор условий, при которых объект имеет смысл.
- 7 Подкласс не должен изменять значения приватных полей базового класса.

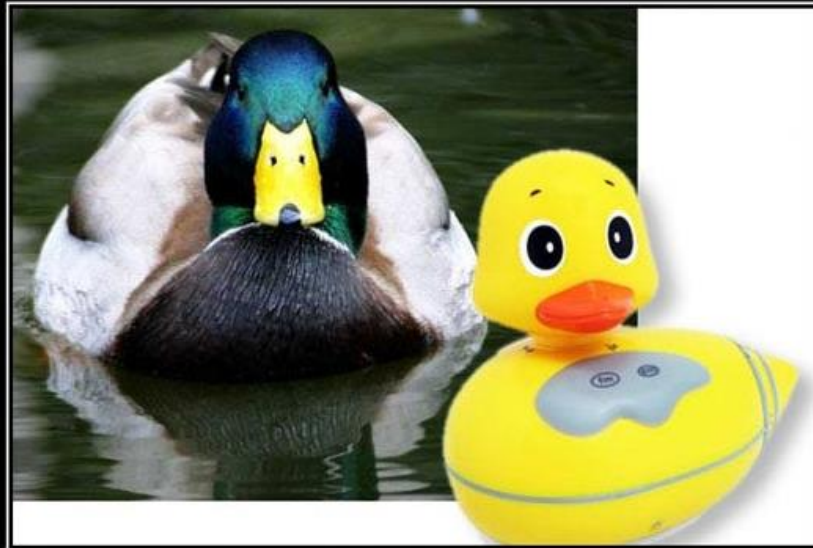
Солидный, крепкий, надёжный

Liskov substitution principle



LSKOV SUBSTITUTION PRINCIPLE

No matter what you used to learn how to drive, you should be able to drive any car afterward.



LSKOV SUBSTITUTION PRINCIPLE

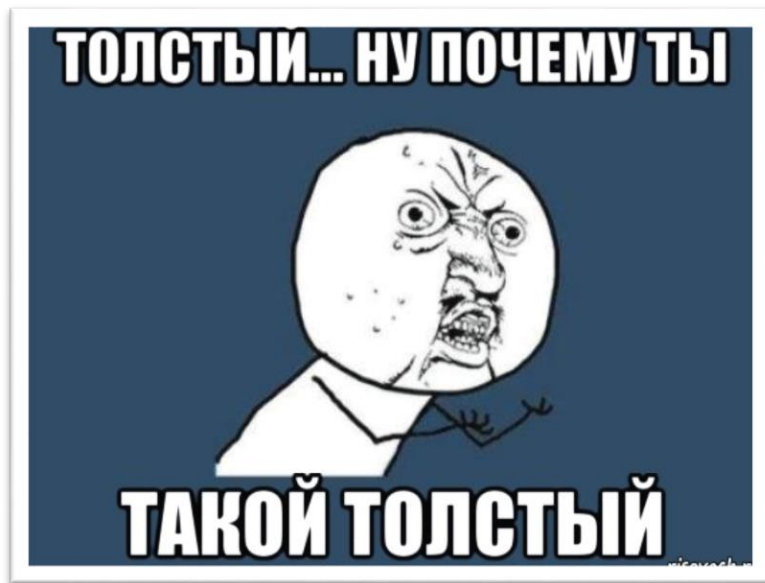
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Солидный, крепкий, надёжный

Interface segregation principle

Принцип разделения интерфейса – много специализированных интерфейсов лучше, чем один универсальный.

Клиенты не должны зависеть от методов, которые они не используют, поэтому следует избегать слишком «толстых» интерфейсов.



Солидный, крепкий, надёжный

Dependency inversion principle

Принцип инверсии зависимостей – зависимости внутри системы строятся на основе абстракций. Модули верхнего уровня не зависят от модулей нижнего уровня. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

- Классы нижнего уровня реализуют базовые операции вроде работы с диском, передачи данных по сети, подключения к базе данных и прочее.
- Классы высокого уровня содержат сложную бизнес-логику программы, которая опирается на классы низкого уровня для осуществления более простых операций.



Солидный, крепкий, надёжный

Dependency inversion principle

Зачастую вы сперва проектируете классы нижнего уровня, а только потом берётесь за верхний уровень. При таком подходе классы бизнес-логики становятся зависимыми от более примитивных низкоуровневых классов. Каждое изменение в низкоуровневом классе может затронуть классы бизнес-логики, которые его используют.

Принцип инверсии зависимостей предлагает изменить направление, в котором происходит проектирование.



Солидный, крепкий, надёжный

Dependency inversion principle



Порядок применения принципа инверсии зависимостей:

- 1 Для начала нужно описать интерфейс низкоуровневых операций, которые нужны классу бизнес-логики.
- 2 Это позволит вам убрать зависимость класса бизнес-логики от конкретного низкоуровневого класса, заменив её «мягкой» зависимостью от интерфейса.
- 3 Низкоуровневый класс, в свою очередь, будет реализовывать интерфейс.

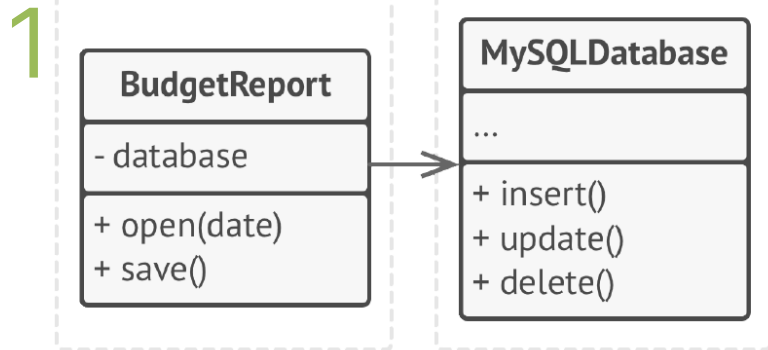
Солидный, крепкий, надёжный

Dependency inversion principle

Высокоуровневый класс зависит от
низкоуровневого (использует его)

Высокий уровень

Низкий уровень



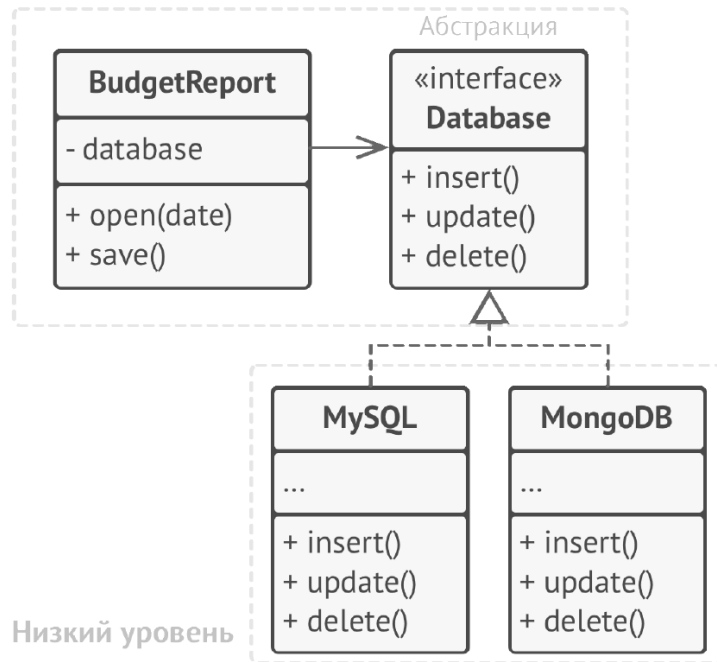
Высокоуровневый класс использует абстракцию.

Низкоуровневый класс зависит от абстракции

Высокий уровень

Абстракция

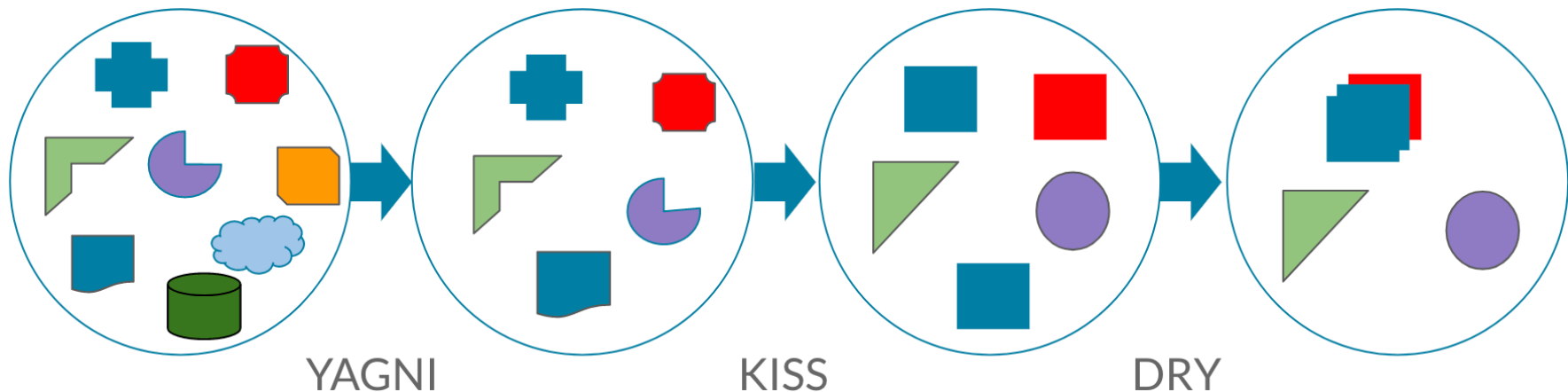
2



И всё?

Совмещение принципов

Чаще всего принципы сосуществуют мирно и дополняют друг друга. Тем не менее, иногда бывают ситуации, когда принципы начинают противоречить друг другу. В таком случае в первую очередь придерживайтесь *SOLID* и тех принципов, что ему не противоречат в данной ситуации. Остальные принципы игнорируйте.

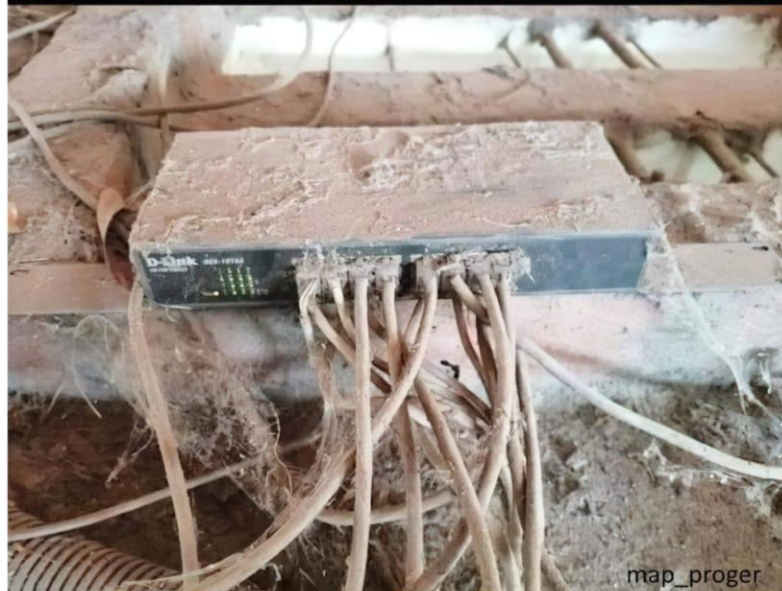


И всё?

Другие практики



Если работает – не трогай



И всё?

Другие практики

Когда ты удаляешь код, который тебе кажется ненужным



По шаблонам

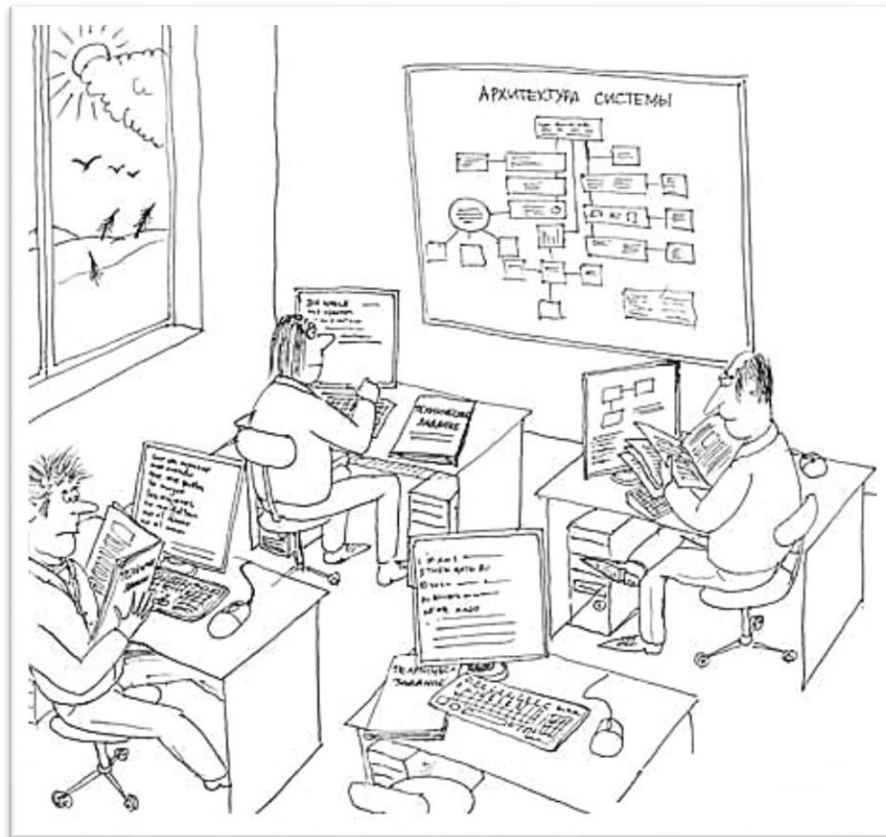
Паттерны проектирования

Паттерн (шаблон) проектирования - это часто встречающееся решение определённой проблемы при проектировании архитектуры программ. В отличие от готовых функций или библиотек, паттерн нельзя просто взять и скопировать в программу.

Паттерн != Алгоритм, т. к.

Паттерн – общее описание решения, само решение может отличаться в каждом случае

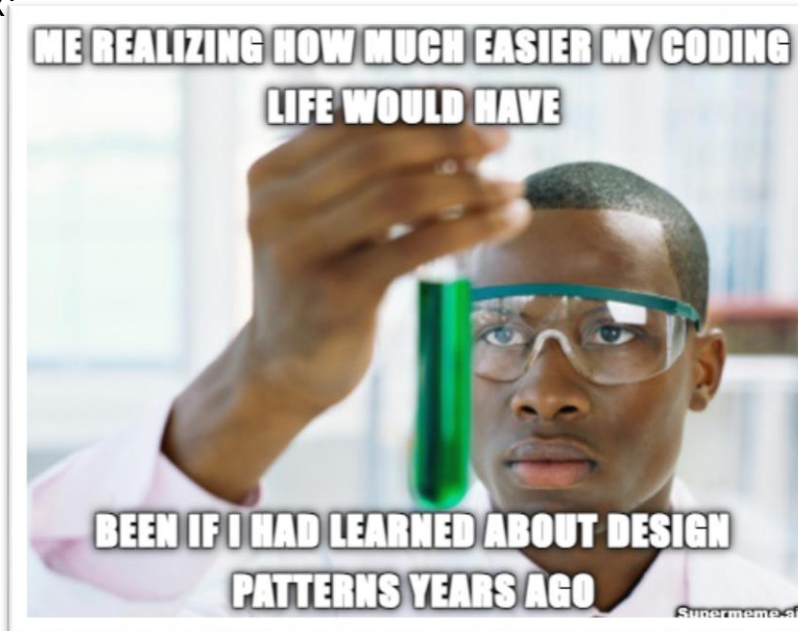
Алгоритм – чёткий набор действий.



По шаблонам

Структура паттерна

- проблема, которую решает паттерн;
- структуры классов, составляющих решение;
- пример на одном из языков программирования;
- особенности реализации в различных контекстах;
- связи с другими паттернами.



По шаблонам

Паттерны, которые встречали



[Легковес](#) – паттерн, который помогает экономить память при создании строк (механизм пула строк).

[Прототип](#) – паттерн, который помогает клонировать объекты (применяли при создании клонирующего конструктора)

[Итератор](#)* – паттерн, предполагающий вынос логики обхода коллекции в отдельный класс.

[Строитель](#)* – паттерн последовательной инициализации сложного объекта (используется в StringBuilder)

* часто реализуется внутренним классом



prg_memes

Если вы первым делом подумали "что-то о программировании", вы такой же поехавший, как и я.



По шаблонам

Другие паттерны

Другие паттерны можно изучать здесь

<https://refactoring.guru/ru/design-patterns/catalog>

или прочитать книгу «Погружения в паттерны проектирования» (А. Швец)



Задание

1 Создайте класс Форма регистрации с 10 полями. Пользователь заполняет в форме только те поля, которые хочет.

2 Реализуйте паттерн *Строитель* с помощью внутреннего класса, позволяющий создавать объект формы.

<https://refactoring.guru/ru/design-patterns/builder>



3

Домашнее задание

Домашнее задание

1 Напишите примеры классов/интерфейсов/отдельных методов с нарушением изученных принципов (4-5 нарушений). В комментариях, начинающихся с TODO, укажите, какой принцип был нарушен. Получившийся проект направьте на проверку.

2 Создайте класс Пицца с минимум 5 полями. Реализуйте внутренний класс-строитель, помогающий компоновать объект Пицца.



Дополнительная практика

Изучите паттерн Наблюдатель <https://refactoring.guru/ru/design-patterns/observer>

Создайте приложение для оповещения граждан об опасности. Создайте классы разных типов граждан (обычные, службы спасения, администрации учреждений, армия и полиция).

В зависимости от типа опасности система должна оповещать разные категории :

- Природная катастрофа – все граждане по всем каналам.
- Техногенная катастрофа – службы спасения, администрации учреждений
- Уличные беспорядки – армия и полиция
- Тестирование системы оповещения – обычные граждане, администрации

ЗАКЛЮЧЕНИЕ

