
<TeachMeSkills/>

курс

Android разработчик

Занятие 25. Основные принципы программирования

Агенда занятия

SOLID
DRY
KISS
YAGNI
Unidirectional data flow

- **SOLID**
  - DRY
  - KISS
  - YAGNI
  - Unidirectional data flow

Что такое SOLID?

SOLID — это аббревиатура пяти основных принципов объектно-ориентированного проектирования, которые помогают писать гибкий, поддерживаемый и тестируемый код. Принципы особенно актуальны для Android-разработки, где часто приходится работать с большими и сложными приложениями.

Принципы SOLID

S — Single Responsibility Principle (Принцип единственной ответственности)

Класс должен иметь только одну причину для изменения и иметь одну зону ответственности.

Принципы SOLID

Пример: Вместо одного класса, который загружает данные из сети и отображает их, раздели ответственность

```
// No Single Responsibility Principle (Принцип единственной ответственности)
class NetworkRepositoryBad {
    fun fetchData(): String {
        // Получение данных из сети
        return "Data from network".uppercase(Locale.getDefault()).filter { it.isUpperCase() }
            .slice( indices: 1 ≤ .. ≤ 3)
    }
}
```

```
class NetworkRepository {
    fun fetchData(): String {
        // Получение данных из сети
        return "Data from network"
    }
}

class DataViewModel(private val repository: NetworkRepository) {
    fun getData() {
        val data = repository.fetchData()
        // Обработка данных
    }
}
```

Принципы SOLID

O — Open/Closed Principle (Принцип открытости/закрытости)

Классы должны быть открыты для расширения, но закрыты для модификации.

Пример: Вместо изменения класса, используем наследование или интерфейсы

```
class Shape {
    fun draw(type: String) {
        if (type == "circle") {
            // draw circle
        } else if (type == "rectangle") {
            // draw rectangle
        }
    }
}
```

```
interface Logger {
    fun log(message: String)
}

class ConsoleLogger : Logger {
    override fun log(message: String) {
        println(message)
    }
}

class FileLogger : Logger {
    override fun log(message: String) {
        // Логирование в файл
    }
}
```

Принципы SOLID

L — Liskov Substitution Principle (Принцип подстановки Лисков)

Объекты подклассов должны заменять объекты базового класса без нарушения работы программы.

```
open class Bird {
    open fun fly() { /* ... */ }
}

class Ostrich : Bird() {
    override fun fly() {
        throw UnsupportedOperationException("Ostrich can't fly")
    }
}
```

```
open class Animal {
    open fun makeSound() {}
}

class Dog : Animal() {
    override fun makeSound() { println("Bark") }
}

class Cat : Animal() {
    override fun makeSound() { println("Meow") }
}

fun playSound(animal: Animal) {
    animal.makeSound()
}
```

Принципы SOLID

I — Interface Segregation Principle (Принцип разделения интерфейса)

Лучше много специализированных интерфейсов, чем один общий.

```
interface Worker {
    fun work()
    fun eat()
}

class Robot : Worker {
    override fun work() {}
    override fun eat() {} // Не нужен для робота!
}
```

```
interface Workable {
    fun work()
}

interface Eatable {
    fun eat()
}

class RobotGood : Workable {
    override fun work() {}
}

class Human : Workable, Eatable {
    override fun work() {};
    override fun eat() {}
}
```

Принципы SOLID

D — Dependency Inversion Principle (Принцип инверсии зависимостей)
Модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба должны зависеть от абстракций.

```
class UserRepository {
    private val networkService = NetworkService()
    // ...
}

class NetworkService {
    fun fetchData() = "Data from network"
}
```

```
class UserRepositoryGood(private val networkService: INetworkService)

interface INetworkService {
    fun fetchData(): String
}
```

SOLID

▶ **DRY**

KISS

YAGNI

Unidirectional data flow

Что такое DRY?.

DRY (Don't Repeat Yourself) — «Не повторяйся».
Принцип призывает избегать дублирования кода и знаний, чтобы облегчить поддержку, развитие и тестирование приложения.

К чему приводит нарушение DRY:
- Сложнее исправлять баги (нужно менять один и тот же код в нескольких местах).
- Труднее вносить новые функции (из-за копипасты можно что-то забыть изменить).
- Больше вероятность ошибок и рассинхронизации логики.

Примеры нарушения DRY и их исправление

```
button1.setOnClickListener {
    Toast.makeText( context: this,  text: "Button 1 clicked", Toast.LENGTH_SHORT).show()
}
button2.setOnClickListener {
    Toast.makeText( context: this,  text: "Button 2 clicked", Toast.LENGTH_SHORT).show()
}
```

```
    button1.setOnClickListener { showToast("Button 1 clicked") }
    button2.setOnClickListener { showToast("Button 2 clicked") }
}


fun showToast(message: String) {
    Toast.makeText( context: this, message, Toast.LENGTH_SHORT).show()
}
```

Примеры нарушения DRY и их исправление

Повторяющиеся layout-элементы

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<include layout="@layout/profile_block" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Utility функции

```
fun String.isEmailValid() =
    android.util.Patterns.EMAIL_ADDRESS.matcher( input: this).matches()

fun validateFields() {
    val email = "email"
    email.isEmailValid()
}
```

SOLID

DRY

▶ **KISS**

YAGNI

Unidirectional data flow

Что такое KISS?

KISS (Keep It Simple, Stupid)
В переводе: "Делай проще, глупец" (или мягче — "Делай проще!").

Смысл:
Системы и решения должны быть максимально простыми, избегай излишней сложности.
"Если можно сделать просто — делай просто!"

Как KISS проявляется в Android-разработке?

- Не усложняй архитектуру, если задача элементарная.
- Используй стандартные компоненты и паттерны.
- Пиши читаемый, понятный код.
- Не внедряй абстракции и библиотеки без необходимости.
- Минимизируй количество зависимостей.

Примеры KISS

```
    button.setOnClickListener {
        Toast.makeText( context: this,  text: "Clicked!", Toast.LENGTH_SHORT).show()
    }

    button.setOnClickListener{
        MyButtonClickHandler().onClick()
    }
}
```

```
interface ClickHandler {
    fun onClick()
}

class MyButtonClickHandler : ClickHandler {
    override fun onClick() { /* ... */ }
}
```

SOLID

DRY

KISS

▶ **YAGNI**

Unidirectional data flow

Что такое YAGNI?

YAGNI (You Aren't Gonna Need It) — «Тебе это не понадобится».

Суть:
Не реализуй функционал и абстракции, которые "вдруг пригодятся в будущем", если в них нет явной текущей необходимости.
Иначе: не пиши код, который не требуется для решения текущей задачи.

Почему YAGNI важен для Android-разработки?

- Экономия времени и сил: ненужные фичи — лишние строки, тесты и баги.
- Поддерживаемость: меньше кода — проще читать, поддерживать, обучать коллег.
- Производительность: меньше зависимостей, быстрее сборка, меньше размер APK.
- Фокус: команда занимается только тем, что реально востребовано.

Что такое YAGNI?

```
// В приложении пока один тип авторизации, но разработчик сразу делает абстракцию "на будущее"
interface AuthStrategy {
    fun authenticate()
}

class EmailAuthStrategy : AuthStrategy {
    override fun authenticate() { /* ... */
    }
}
```

```
// Хотя сейчас только EmailAuthStrategy используется


// Делай просто, пока реально нужен только Email
class AuthManager {
    fun authenticateWithEmail(email: String, password: String) { /* ... */
    }
}
```

SOLID

DRY

KISS

YAGNI

- **Unidirectional data flow**

Unidirectional Data Flow

Unidirectional Data Flow — паттерн проектирования, при котором данные в приложении всегда движутся в одном направлении (по заранее определённому циклу), а не "скачут" туда-сюда между компонентами.
Главная идея:
- Единый источник истины (source of truth) для состояния. - Изменения состояния происходят централизованно (например, через события, actions). - View только отображает состояние и отправляет действия. - Нет сложных и неконтролируемых обратных связей.

Зачем нужен UDF?
Предсказуемость: всегда понятно, откуда и как изменяется состояние.
Тестируемость: логику легче тестировать — меньше сайд-эффектов.
Проще отлаживать: проще найти причину изменения данных.
Масштабируемость: хорошо подходит для сложных UI.

Unidirectional Data Flow

Смотрим проект.

Практика

Задачи

Задача 1: Исправить соблюдая принципы SOLID

Ссылка на Git.

Задачи

Задача 2: Исправить соблюдая принцип DRY, KISS, YAGNI

Ссылка на Git.

Задачи

Задача 3: Реализуйте UnitDirectionalDataFlow используя ViewModel

Ссылка на Git.

Q&A

Домашнее задание

Задачи

Задача 1: Clean Architecture + SOLID + DRY + YAGNI + KISS

Реализуйте простой экран "Список покупок".

Требования:

1. Пользователь видит список покупок (названия товаров).
2. Пользователь может добавить новый товар через поле ввода и кнопку "Добавить".
3. Пользователь может отметить товар как купленный (чекбокс).
4. Состояние списка сохраняется только в памяти
5. Экран реализуйте в одном Activity или Fragment.
6. Используйте RecyclerView для списка.

Задачи

Задача 2*: Дизайн система

Реализуйте все компоненты задания 1 как элементы дизайн системы и используйте их. Не создавая новых элементов.
Например текст с настроенным шрифтом, размером. Все кнопки одинаковые.

Q&A

Ваши вопросы

Спасибо