Лекция 5 - Архитектура приложения и продвинутые паттерны проектирования

Тлеубаева А.О.

Содержание

- • MVVM, MVP, MVI: Определение и сравнение
- • Dependency Injection: Зачем и как
- • Тестирование: Unit и UI тестирование

MVVM

- Model: Бизнес-логика и обработка данных (БД, сетевые запросы).

Model в архитектуре MVVM отвечает за бизнес-логику и управление данными. Это может включать в себя работу с базами данных, сетевые запросы, а также обработку и хранение данных.

Пример: Представьте, что у нас есть мобильное приложение для чтения новостей. Model будет взаимодействовать с удаленным сервером для получения новостей и может кэшировать их в локальной базе данных для быстрого доступа.

MVVM

- View: Отображение UI, отвечает только за визуальное представление данных.

Определение: View отвечает за отображение данных пользователю и взаимодействует только с ViewModel, чтобы получить необходимые данные.

Пример: В нашем примере с приложением для чтения новостей, View может представлять собой экран со списком новостных статей. Он отобразит статьи, полученные от ViewModel, и передаст действия пользователя (например, выбор статьи) обратно в ViewModel.

MVVM

- ViewModel: Связующий элемент между Model и View, обрабатывает данные и логику отображения.

ViewModel действует как связующее звено между Model и View. Она получает данные от Model, обрабатывает их (если необходимо) и передает готовые для отображения данные в View. Кроме того, ViewModel принимает ввод от View и выполняет соответствующие действия, которые могут включать обновление Model.

- **Пример**: ViewModel в нашем приложении для чтения новостей будет получать новости от Model, возможно, форматировать их для удобного отображения и передавать в View для показа пользователю. При выборе статьи пользователем ViewModel может также управлять навигацией к экрану детализации статьи.

Демонстрация MVVM в мобильной разработке

- **Data Binding**: Один из ключевых аспектов MVVM — это двусторонний data binding между View и ViewModel, который автоматически обновляет UI, когда изменяются данные, и наоборот. Это уменьшает объем шаблонного кода во View и позволяет сосредоточиться на бизнес-логике.
- **Тестирование**: MVVM поддерживает разделение ответственности, что делает тестирование более простым. ViewModel может быть легко протестирована независимо от UI и базы данных.
- **Переиспользование кода**: ViewModel не имеет прямой зависимости от View, что делает возможным повторное использование кода ViewModel в различных частях приложения.

MVP

- Model: Обработка данных и бизнес-логики.
- Model в архитектуре MVP отвечает за бизнес-логику и управление данными приложения. Это может включать в себя сетевые запросы, работу с базой данных и любую другую логику обработки данных.
- View: Отображение данных, передача действий пользователя в Presenter.
- Presenter: Принимает ввод от View, работает с Model и обновляет View.
- **Пример**: В контексте мобильного приложения для заказа еды, Model может взаимодействовать с API ресторана для получения меню и обрабатывать заказы пользователя.

MVI

MVI (Model-View-Intent) - это архитектурный паттерн, который был вдохновлен функциональным реактивным программированием и предназначен для облегчения работы с пользовательским интерфейсом в программировании.

- Model: Отвечает за данные и бизнес-логику.
- View: Отображение данных и генерация событий от действий пользователя.
- Intent: Преобразует события пользователя в действия для обновления Model или View.
- **Пример:**
- Допустим, пользователь хочет создать новую заметку в приложении.

1. Пользователь нажимает на кнопку "Создать заметку" в интерфейсе.
2. View генерирует событие создания новой заметки.
3. Intent принимает это событие и преобразует его в действие для Model.
4. Model обрабатывает это действие, добавляя новую заметку в список заметок.
5. Model отправляет обновленные данные обратно в View.
6. View отображает обновленный список заметок, включая новую заметку, которую создал пользователь.

Таким образом, MVI помогает организовать взаимодействие между пользовательским интерфейсом и бизнес-логикой в систематизированной и предсказуемой манере, делая код более чистым и легким для понимания и отладки.

Сравнение паттернов

- **Сложность реализации:**

1. **MVVM**:
    1. Средняя сложность: MVVM обеспечивает хорошее разделение ответственности между компонентами, что облегчает понимание и реализацию.
    2. Поддерживается многими фреймворками, что уменьшает сложность реализации.
2. **MVP**:
    1. Средняя сложность: MVP требует строгого разделения между моделью, представлением и презентером, что может увеличить сложность реализации.
3. **MVI**:
    1. Высокая сложность: MVI представляет собой функциональный подход, который может быть сложным для понимания и реализации, особенно для разработчиков, не знакомых с реактивным программированием.

Сравнение паттернов

- **Тестирование:**

1. **MVVM**:
    1. Легкость тестирования: благодаря четкому разделению ответственности и поддержке различных библиотек.
2. **MVP**:
    1. Легкость тестирования: также благодаря четкому разделению между компонентами.
3. **MVI**:
    1. Тестирование может быть сложным из-за реактивного характера этого паттерна.

- **Управление состоянием:**

1. **MVVM**:
    1. Управление состоянием может быть сложным из-за двусторонней привязки данных.
2. **MVP**:
    1. Управление состоянием в основном зависит от презентера, что может создать дополнительные сложности.
3. **MVI**:
    1. Централизованное управление состоянием, что обеспечивает предсказуемость и упрощает отладку.

Сравнение паттернов

- **Примеры и сценарии использования:**

1. **MVVM**:
    1. Примеры: Приложения на базе XAML, такие как WPF, UWP или Xamarin.Forms.
    2. Сценарии: Приложения с сложным пользовательским интерфейсом и взаимодействием с пользователем.
2. **MVP**:
    1. Примеры: Android приложения с использованием GWT.
    2. Сценарии: Приложения с простым пользовательским интерфейсом и строгим разделением между логикой представления и модели.
3. **MVI**:
    1. Примеры: Приложения, использующие библиотеки, такие как Redux или MobX.
    2. Сценарии: Приложения с сложным управлением состоянием и реактивным программированием.

- Выбор между MVVM, MVP и MVI в конечном итоге зависит от конкретного проекта, команды разработчиков и технологического стека.

Dependency Injection

Обзор принципов Dependency Injection и их важности в разработке.

- **Принципы Dependency Injection (DI) и их важность:**
- Dependency Injection (DI) является ключевым паттерном проектирования, который используется для уменьшения степени связанности между компонентами программы. Основные принципы и важность DI включают в себя:

1. **Разделение ответственности**: DI помогает разделить ответственность за создание зависимостей и их использование, что делает систему более модульной и легко тестируемой.
2. **Повторное использование кода**: Уменьшив связанность, можно легче повторно использовать и заменять компоненты системы.
3. **Повышение тестируемости**: DI облегчает замену реальных зависимостей макетами или фиктивными объектами во время тестирования.
4. **Конфигурация на уровне кода**: Вместо жесткого кодирования конкретных реализаций, DI позволяет настраивать систему динамически через внешние конфигурации.

Dagger и Koin в Android-проектах

- Dagger — это популярная библиотека DI для Java и Android, которая использует аннотации для генерации кода, управляющего зависимостями.
- Koin — это легкая библиотека DI для Kotlin, которая не требует использования аннотаций и предлагает DSL для описания зависимостей.

Обе библиотеки предлагают различные подходы к реализации DI, и выбор между ними будет зависеть от предпочтений команды и требований проекта. Dagger предлагает более строгий и мощный способ управления зависимостями, в то время как Koin предлагает более простой и идиоматический для Kotlin подход.

Введение в тестирование

- **Важность тестирования в обеспечении качества ПО и производительности разработки:**

1. **Обеспечение качества**: Тестирование помогает обеспечить, что ваше ПО работает корректно и свободно от багов. Это обеспечивает уверенность в том, что программное обеспечение выполняет свои функции в соответствии с требованиями.
2. **Уменьшение рисков**: Тестирование помогает выявить и исправить ошибки на раннем этапе разработки, что снижает риски отказа ПО в будущем.
3. **Повышение производительности разработки**: Автоматизированное тестирование может ускорить процесс разработки, обеспечивая быструю обратную связь разработчикам о состоянии их кода.
4. **Повышение удовлетворенности клиентов**: ПО высокого качества, прошедшего тщательное тестирование, скорее всего, приведет к удовлетворенности клиентов, что в свою очередь повысит репутацию вашей компании на рынке.

Введение в тестирование

- **Обзор видов тестирования: Unit и UI тестирование:**

Unit тестирование:

1. **Цель**: Проверка отдельных единиц кода (например, функций, методов) на корректность.
2. **Инструменты**: JUnit, Mockito, Espresso для Android, XCTest для iOS.
3. **Преимущества**: Быстрое исполнение, выявление ошибок на раннем этапе, упрощение процесса рефакторинга.
4. **Пример**:

```
@Test
public void addition_isCorrect() {
assertEquals(4, 2 + 2);
}
```

Введение в тестирование

- **UI тестирование**:
- **Цель**: Проверка интерфейса пользователя на соответствие требованиям и корректную работу в различных условиях.
- **Инструменты**: Selenium, Appium, Espresso для Android, XCUITest для iOS.
- **Преимущества**: Позволяет обеспечить корректное взаимодействие пользователя с приложением, обеспечивает тестирование в реальных условиях использования.
- **Пример** (используя Espresso для Android):

```
@Test
public void buttonClickNavigatesToDetail() {
onView(withId(R.id.button)).perform(click());
onView(withId(R.id.detail_view)).check(matches(isDisplayed()));
}
```

В зависимости от проекта и команды разработки, эти виды тестирования могут быть включены в процесс разработки в различных комбинациях для обеспечения высокого качества и производительности ПО.

Контрольные вопросы

- **Архитектурные паттерны**

1. Что такое архитектурный паттерн в разработке ПО и какова его роль?
2. Опишите основные компоненты и принципы работы паттерна MVVM.
3. Какие преимущества и недостатки у паттерна MVP по сравнению с MVVM?
4. В чем основная идея паттерна MVI и как он отличается от MVVM и MVP?
5. Как архитектурные паттерны влияют на тестирование приложения?

- **Dependency Injection**

6. Что такое Dependency Injection и каковы его основные преимущества?
7. Какие существуют подходы к реализации Dependency Injection?
8. В чем различия между Dagger и Koin при реализации DI в проектах Android?

- **Тестирование**

9. Почему тестирование важно в процессе разработки ПО?
10. Какие основные типы тестирования применяются в разработке мобильных приложений?
11. Что такое Unit-тестирование и какие его основные принципы?
12. Какие инструменты используются для UI тестирования в Android и iOS разработке?
13. Какие вызовы могут возникнуть при написании UI тестов и как их минимизировать?

- **Общие вопросы**

14. Как вы бы выбрали архитектурный паттерн для нового проекта и на основе каких критериев?
15. Какие подходы вы используете, чтобы обеспечить высокое качество кода в вашем проекте?

Практические задачи

1. Разработайте простой экран с использованием паттерна MVVM (или MVP, MVI).
2. Напишите пример Unit-теста для тестирования конкретной функции или метода.

3. Приведите пример, как вы бы реализовали Dependency Injection в конкретном классе или модуле.