Android App
Architecture, Testing
Architectural principles

• One crucial aspect of Android applications is that application components can be started/stopped in a variety of ways

For example:
  • Your app starts another app (Camera)
  • After taking a picture, the flow returns to your app
  • But then, the user gets a phone call, a friend is inviting them to lunch
  • Now, the user puts your app in the background and starts searching for lunch options instead..
  • A few hours later, the user finally comes back to your app and wants to finish the task they started earlier

• Since you don’t have control over such interruptions, app data / state shouldn’t be stored in the components
Principles: Separation of concerns

• You should aim for separation of concerns
  • i.e. don’t just write all your code in an Activity or Fragment..

• UI classes should be lean and only deal with UI-related logic – displaying data or fetching user input
  • Try to minimize your dependency on them
Principles: Model-driven UI

• Model – component that is responsible for handling the data
  • Ideally, based on persistent storage

• Independent from the View objects, app components

By designing our code using model classes that have clear responsibility of managing the data, the app becomes easier to test and maintain
General recommended architecture- MVVM

• MVVM : Model-View-ViewModel
• Let’s discuss how to structure our code for a simple example
• The Plant Viewer App
  • Can read plants from a web database
• We will focus on just the Plant Details view – show a plant’s details, knowing its ID
View, ViewModel

• “Plant Details” View:
  • Activity/Fragment – UI controller which displays the data
  • XML layout file defines the UI for the screen

• Data for the plant details view:
  • Plant ID
    • The bare minimum to restore state
  • Plant object data class
    • The details about the plant: genus, title, etc.

• This data is kept in a ViewModel
  • Manages data for a specific UI component
  • Calls other components to load data
  • Mediates user requests to modify data
  • Doesn’t directly know anything about the UI components (e.g. not concerned with orientation changes)
LiveData

• When the Fragment is launched, consider it gets a **Plant ID** as a Fragment argument
• The ViewModel should now load the data corresponding to the ID
• The Fragment and ViewModel are connected through **LiveData** objects
  • LiveData – an observable data holder
• After loading the data, the Plant Object is updated using LiveData and the UI controller is notified via the observer callback
Architecture – Loading UI

• How does ViewModel fetch the plant data?
  • Imagine we use some web back-end API
  • We could directly invoke the network API requests in the ViewModel, and update our LiveData.
  • This is not well-aligned with separation of concerns principle

• Instead, let’s add an additional module - Repository

https://developer.android.com/jetpack/guide#fetch-data
Repository

• **Repository** – custom class dedicated to data operations

• Should provide a clean API to abstract away data sources from rest of the app
  • ViewModel need not be concerned that a web API is used

• A mediator between different data sources - e.g. web service + cache

• The repository could provide a method such as:
  • `fun getPlant(plantId: String): Plant`

https://developer.android.com/jetpack/guide#fetch-data
Repository

- **Repository** – custom class dedicated to data operations
- Should provide a clean API to abstract away data sources from rest of the app
  - ViewModel need not be concerned that a web API is used
- A mediator between different data sources – e.g. web service + cache
- The repository could provide a method such as:
  - `fun getPlant(plantId: String): Plant`

https://developer.android.com/jetpack/guide#fetch-data
Dependency Injection

• A common issue is how to manage references to instances of objects shared between modules.
  • E.g., multiple ViewModels may depend upon the PlantRepository
  • A common solution is **Dependency Injection**
    • Classes declare classes the depend upon, but don’t construct them
    • Another library class provides the dependencies during runtime
    • **Hilt library** (based on Dagger)

https://developer.android.com/jetpack/guide#fetch-data
Dependency Injection

• Example with Hilt:
  • PlantViewModel declares PlantRepository as a dependency using @ViewModelInject

```kotlin
class PlantViewModel @ViewModelInject constructor(
    plantRepository: PlantRepository
) : ViewModel() {
    val plantId : String = ...
    val plant : LiveData<Plant> =
        MutableLiveData(plantRepository.getPlant(plantId))
}
```

https://developer.android.com/jetpack/guide#fetch-data
Cacheing

- Right now, the Repository fetches the data, and it’s displayed in the UI, but not stored anywhere.
- As a result, if the user briefly closes the app and re-opens it, the data must be re-fetched.
  - This wastes network bandwidth
  - User has to wait for network request completion
- To solve this, we could add another datasource

https://developer.android.com/jetpack/guide#fetch-data
Cacheing

- We could use a local Room DB for persistent storage and cacheing

- Then, `getPlant()` would:
  1. Check if there’s a cached version for the provided plant ID
  2. If yes, return the cached version
  3. If no, perform the network request, retrieve the result, add it to the cache / DB and return it.

https://developer.android.com/jetpack/guide#fetch-data
Single source of truth

• If you configure your Room DB DAO-s to return LiveData type objects, you can:
  • Have the Repository effectively save web service results to the database
  • Changes to the database trigger updates in the UI
• This way, the database serves as the single source of truth
• See also: *Flow for Room*

https://developer.android.com/jetpack/guide#truth
https://developer.android.com/training/data-storage/room/async-queries#observable
Architecture summary

• The described architectural design is called Model-View-ViewModel (MVVM)
• Each component depends only on the component one level below it.
• Helps improve maintainability of your project, e.g. easier testing
• Better user experience:
  • Persistent storage, avoiding making too many web requests, etc
Architecture

• Alongside MVVM, you may have heard of MVC, MVP
• MVC – Controller trends to get bloated, is heavily linked to View aspects
• MVP – Better than MVC, but needs extra effort of maintaining interfaces (Presenter updates View through interface)
Android Architecture Blueprints

- [https://github.com/android/architecture-samples/](https://github.com/android/architecture-samples/)
  - Examples of architectural approaches
- E.g. Another modern pattern is the **Single-Activity architecture**
  [https://www.youtube.com/watch?v=2k8x8V77CrU](https://www.youtube.com/watch?v=2k8x8V77CrU)
Testing Android Apps
Testing Introduction

• How to write perfect code?

• Are you familiar with unit testing in Java / Javascript / .NET / ... ?
Testing - Fundamentals

• Separate responsibilities into modular units
  • Iterative process, you may have to (re-)think about responsibilities with every new feature!

• For each unit, write a set of unit tests
  • Should be exhaustive
    • Consider all interactions
    • Corner cases
    • Consider possible inputs (incl. invalid ones!)

• Validate integration between units with UI tests & Integration tests
  • Flows & interactions which encompass several units

https://developer.android.com/training/testing/fundamentals
Testing Pyramid

<table>
<thead>
<tr>
<th>Test Type</th>
<th>Validates interactions...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small Tests</td>
<td>..of a single unit, one class at a time</td>
</tr>
<tr>
<td>(Unit tests)</td>
<td></td>
</tr>
<tr>
<td>Medium Tests</td>
<td>a) ..between levels of stack within a module</td>
</tr>
<tr>
<td>(Integration tests)</td>
<td>b) ..between related modules</td>
</tr>
<tr>
<td>Large Tests</td>
<td>..of user journeys through multiple app modules</td>
</tr>
<tr>
<td>(UI tests)</td>
<td></td>
</tr>
</tbody>
</table>

https://developer.android.com/training/testing/fundamentals
Unit tests

*Fundamental tests in a software testing strategy*

- Test smallest possible units of code
  - Class, method, component

- Exhaustively validate the functionality and contracts of each class within your app.

- Units usually tested in isolation
  - Affect and monitor changes only to the unit under test
  - Sometimes, isolating an unit is not straight-forward:
    - Mock the behaviour of the components which unit depends on
Typical Java Unit testing

JUnit framework offers convenient ways to perform reusable setup, teardown and assertion operations

```java
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculatorTest {

    @Test
    public void evaluatesExpression() {
        Calculator calculator = new Calculator();
        int sum = calculator.evaluate("1+2+3");
        assertEquals(6, sum);
    }
}
```

https://developer.android.com/training/testing/start/index.html#junit
Android Unit testing

There are two types:

1. **Local Unit tests**
   - Run on local JVM
   - Isolate unit from dependencies using mock objects
     - Mocking framework such as *Mockito*
   - No access to Android framework API
     - Can use a framework like *Roboelectric*, a dependency provider

2. **Instrumented tests**
   - Run on an Android device (physical/emulator)
   - For test cases with more complex Android dependencies

https://developer.android.com/training/testing/unit-testing
• JVM execution – **fast!**

• Can’t use Android framework dependencies
  • Can use mock objects instead

• Access to instrumentation info (e.g. app **Context**)

• Use if you can’t easily isolate the framework dependencies
Local Unit Tests

Set-up:
- Test files in `module-name/src/test/java`
- JUnit 4 dependency in your app’s `build.gradle`
  - *(done by default in Android Studio)*

```java
// Required -- JUnit 4 framework
testImplementation 'junit:junit:4.12'

// Optional -- Robolectric environment
testImplementation 'androidx.test:core:1.3.0'

// Optional -- Mockito framework
testImplementation 'org.mockito:mockito-core:1.10.19'
```

Let’s run our first unit test!

https://developer.android.com/training/testing/unit-testing/local-unit-tests.html
Test Subject: Shopping Basket App

2 classes:

• **ShoppingItem**
  - Name, price, quantity
  - Price determined by no. of letters!

• **ShoppingBasket**:  
  - Holds items in a collection  
  - Can add items  
    - re-adding item with same name updates items quantity  
  - Calculates total price of items in basket

<table>
<thead>
<tr>
<th>Name</th>
<th>Price</th>
<th>Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bread</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Egg</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Milk</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>
Demo

```kotlin
class ExampleUnitTest {
    private lateinit var basket: ShoppingBasket

    // Note: "@Before" gets called before each @Test method
    @Before
    fun initEmptyBasket() {
        basket = ShoppingBasket()
    }

    @Test
    fun itemPrice() {
        val item = ShoppingItem("Milk")
        assertEquals(4, item.price)
    }
}
```
@Test
fun problematicUnitTest() {
    val item3 = ShoppingItem("12345")
    // getReformattedName() uses some Android library functions
    item3.getReformattedName()
    assertEquals("Product 12345", item3.name)
}

java.lang.AssertionError: Expected: "Product 12345", but: "Product 12345" is null

at android.text.TextUtils
at android.text.TextUtils
at android.text.TextUtils
at ...
Unit test isolation discussion

• Unit tests are executed using a modified android.jar library, which has empty implementations

• Helps encourage the “test-in-isolation” mindset

So any time you find your test not working because of this, you can either:

1) Rethink your tests- are you really testing independent units?

2) Isolate by using mock objects

3) Use a framework like Roboelectric*

* http://robolectric.org/
Local Unit Testing: Mocking with Mockito
	estImplementation 'org.mockito:mockito-core:1.10.19'

Annotate your test class with:

```java
@RunWith(MockitoJUnitRunner::class)
```

Creating a mock Android object:
- Add the `@Mock` annotation to the field declaration
- Define the behaviour:
  - `when(X).thenReturn(Y)` methods
- Matchers:
  - `anyString()`, `eq()`, ...

https://developer.android.com/training/testing/unit-testing/local-unit-tests.html#mocking-dependencies
```kotlin
@RunWith(MockitoJUnitRunner::class)
class UnitTestSample {

    @Mock
    private lateinit var mockContext: Context

    @Test
    fun readStringFromContext_LocalizedString() {
        // Given a mocked Context injected into the object under test...
        `when`(mockContext.getString(R.string.hello_world))
            .thenReturn(FAKE_STRING)
        val myObjectUnderTest = ClassUnderTest(mockContext)

        // ...when the string is returned from the object under test...
        val result: String = myObjectUnderTest.getHelloWorldString()

        // ...then the result should be the expected one.
        assertThat(result, `is`(FAKE_STRING))
    }
}
```

https://developer.android.com/training/testing/unit-testing/local-unit-tests#mocking-dependencies
Roboelectric

Simulates the runtime for Android API level 16+

- provides community-maintained fakes called *shadows*.
- Run on a JVM-powered development machine
  - This allows you to test code that depends on the framework without using mock objects or an emulator
- Supports the following aspects of the Android platform:
  - Component lifecycles
  - Event loops
  - Resources

http://robolectric.org/

testImplementation "org.robolectric:robolectric:4.4"
Moving on to Instrumented tests...

https://developer.android.com/training/testing/start/index.html#test-types
AndroidX Test libraries & APIs

A set of libraries for testing APIs, including instrumentation and Roboelectric support

Includes:

• AndroidJUnitRunner
• Espresso
• UI Automator

Part of Android Jetpack

https://google.github.io/android-testing-support-library/
Instrumented Unit Tests - setup

• Dependencies:

```java
androidTestImplementation 'androidx.test:runner:1.2.0'
androidTestImplementation 'androidx.test:rules:1.2.0'
```

• Set AndroidJUnitRunner as the default test instrumentation runner:

```java
defaultConfig {
    ...
    testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
}
```

https://developer.android.com/training/testing/unit-testing/instrumented-unit-tests#setup
Instrumented Unit Test Demo

```kotlin
@Test
fun useAppContext() {
    // Context of the app under test.
    val appContext = InstrumentationRegistry.getInstrumentation().targetContext
    assertEquals("ee.ut.cs.shoppingbasket", appContext.packageName)
}
```
Single-App UI testing

Using Espresso
UI testing with Espresso Framework

• Programmatically simulates user interactions
  • Click, swipe, text input, ...
  • Run on Android 2.3.3 (API v10) and up

• Espresso manages synchronization of UI actions
  • Can be a challenge in automated UI testing
Espresso Basics

1. Select a View to work with:
   a) `onView(targetView)` (for Activities)
   b) `onData()` (for AdapterViews)

2. Simulate specific interaction:
   a) `perform(action)`

3. Repeat previous steps to simulate user flow

4. Use `ViewAssertion` methods to verify if actual behaviour matches expected

```java
onView(withId(R.id.my_view))
   .perform(click())
   .check(matches(isDisplayed()));
```

https://developer.android.com/training/testing/ui-testing/espresso-testing.html#build
https://developer.android.com/training/testing/espresso/basics
Espresso is most often used together with **Hamcrest matchers**

**Hamcrest** is a library for declarative style matcher predicates – very useful for testing

```java
// Instead of:
var found = false
for (pet: Pet in petList){
    if (pet == dog) found = true
}
assertEquals(true, found)

assertEquals(testedObject.toDouble(), 42.0)
assertThat(testedObject.toDouble(), `is`(42.0))
```

https://code.google.com/archive/p/hamcrest/wikis/Tutorial.wiki
Espresso ActivityScenarioRule

• JUnit rules:
  • mechanism to enhance tests by running code around a test case execution

• AndroidX test includes some useful JUnit rules to reduce boilerplate code:

• **ActivityScenarioRule**
  • Launches and closes specified activity before each test case

```kotlin
@get:Rule
var activityRule: ActivityScenarioRule<MainActivity>
    = ActivityScenarioRule(MainActivity::class.java)
```

https://developer.android.com/training/testing/junit-rules
Espresso Intents

An extension library to Espresso
- Validate and stub (mock) intents sent out by the app
- Intercept outgoing intents, stub the result, send back the result to the component under test
- focus on your own app's logic while assuming that other apps or the platform will function correctly

androidTestImplementation 'androidx.test.espresso:espresso-intents:3.1.0'

- Initialized in test cases via IntentsTestRule

```java
@get:Rule
var rule: IntentsTestRule<MyActivity> = 
  IntentsTestRule(MyActivity::class.java)
```

https://developer.android.com/training/testing/espresso/intents
Espresso Intents example

```kotlin
@RunWith(AndroidJUnit4::class)
class SimpleIntentTest {

    @get:Rule
    var intentsRule: ActivityScenarioRule<MyActivity> = ActivityScenarioRule(MyActivity::class.java)

    @Before
    fun setUp() {
        Intents.init()
    }

    @After
    fun tearDown() {
        Intents.release()
    }

    @Test
    fun verifyMessageSentToDisplayActivity() {
        // Type a message into a EditText
        onView(withId(R.id.editText_name))
            .perform(typeText(MESSAGE), closeSoftKeyboard())

        // Button should send message to another activity via an explicit intent.
        onView(withId(R.id.button)).perform(click())

        // Verify that DisplayMessageActivity got intent with correct package name and message
        intended(allOf(
            hasComponent(hasShortClassName(".DisplayMessageActivity")),
            toPackage(PACKAGE_NAME),
            hasExtra("EXTRA_KEY", MESSAGE))
    }
```
UI testing with multiple apps

• It’s not uncommon for user flow to span multiple apps
  • HW2 app opened the Camera, and then returned to the original application

• This type of flow can be handled using the **UI Automator** APIs
  • Perform interactions on user and system apps
  • E.g. opening menus and app launchers

---

https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html
UI Automator basic approach

1. Get the **UiDevice** object, this reflects the device being tested
2. Get the **UiObject** object, by calling **findObject()**
3. Simulate user interaction on UiObject, similar to Espresso
   - click(), dragTo(), setText(), clearTextField(), swipeUp(), ..
4. Check that the behaviour matches expected one with Junit asserts

Can also launch intents directly

https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html
UI Automator code sample

```java
// Initialize UiDevice instance
device = UiDevice.getInstance(InstrumentationRegistry.getInstrumentation())

val cancelButton: UiObject = device.findObject(
    UiSelector().text("Cancel").className("android.widget.Button")
)
val okButton: UiObject = device.findObject(
    UiSelector().text("OK").className("android.widget.Button")
)

// Simulate a user-click on the OK button, if found.
if (okButton.exists() && okButton isEnabled) {
    okButton.click()
}
```
Our skills so far..

• Core Topics:
  • App. Lifecycle
  • Components
    • Intents, Activities, Services, Broadcasts, Content Providers
  • Permissions

• Handling Data:
  • User state, Storage, File I/O, Resources

• Constructing UIs:
  • ConstraintLayout, Fragments
  • Bridging Data and UI: Adapters, ViewModel, LiveData

• Background tasks, Testing

https://courses.cs.ut.ee/2021/MCIoT/fall/Main/BestPractices