THE ANDROID TESTING CHEAT SHEET

This testing cheat sheet summarizes the theory content taught in the course. It is by no means a replacement for the course content, but rather serves as something that helps you to recap the complex testing topics.

When working on your own projects after the course, you can use this to quickly understand these concepts again without having to watch the videos again.

WHY DO WE WRITE TEST CASES?

MANUAL TESTING

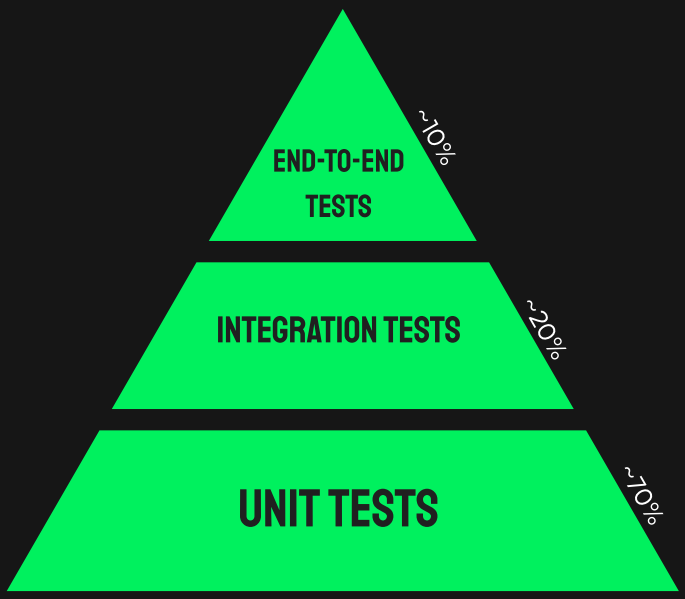
AUTOMATED TESTING

With automated tests,

- ... we avoid breaking older code without noticing
- ... the team has faster feedback loops
- ... we allow our software to scale

TESTS ARE YOUR APP'S LIFE INSURANCE

THE TESTING PYRAMID



See the testing pyramid as a starting point, but don't follow it super stricly

THE TEST RATIOS CAN DIFFER BASED ON THE APP

THE STRUCTURE OF A TEST CASE

```
@Test
fun `Add multiple products, total price sum is correct`() {
    // SETUP
    val product = Product(
        id = 1,
        name = "Ice cream",
        price = 5.0
    )
    cart.addProduct(product, 3)

    // ACTION
    val priceSum = cart.getTotalCost()

    // ASSERTION
    assertThat(priceSum).isEqualTo(15.0)
}
```

1. SETUP

Create class instances and set everything up required for step 2.

2. ACTION

Call the function(s) you want to test.

3. ASSERT

Verify the outcome is what was expected.

JVM VS. INSTRUMENTED TESTS

	RUN ON	SPEED	ACCESS TO ANDROID SDK?	SOURCE SET
JVM TESTS	Java VIRTUAL MACHINE		✗	TEST
INSTRUMENTED TESTS	ANDROID DEVICE		✓	ANDROID TEST

UNIT TESTS

Unit tests test a single isolated unit of code.

- A unit can be seen as a piece of behavior
- Behavior refers to what our code does (and **not** how it does it)
- In code, that's typically a function or a class, but can also be a set of classes

SPEED

LOW

AMOUNT OF COVERED CODE

★★★

MAINTAINABILITY

SUBJECT UNDER TEST

The subject under test refers to the unit or class you're testing.

The test's job is to verify that a part of the subject under test works as intended.

ISOLATION

Isolation means that you eliminate any connections with other parts of the app, so you can test a small piece of code by itself. This is achieved with test doubles.

If a unit test fails, you want to be sure the bug lies in the subject under test.

UNIT → UNIT TEST

INTEGRATION TESTS

Integration tests test the interaction between at least 2 classes.

- They mostly use the real implementations instead of test doubles
- Test doubles may be used for remote APIs to avoid slow and flaky tests

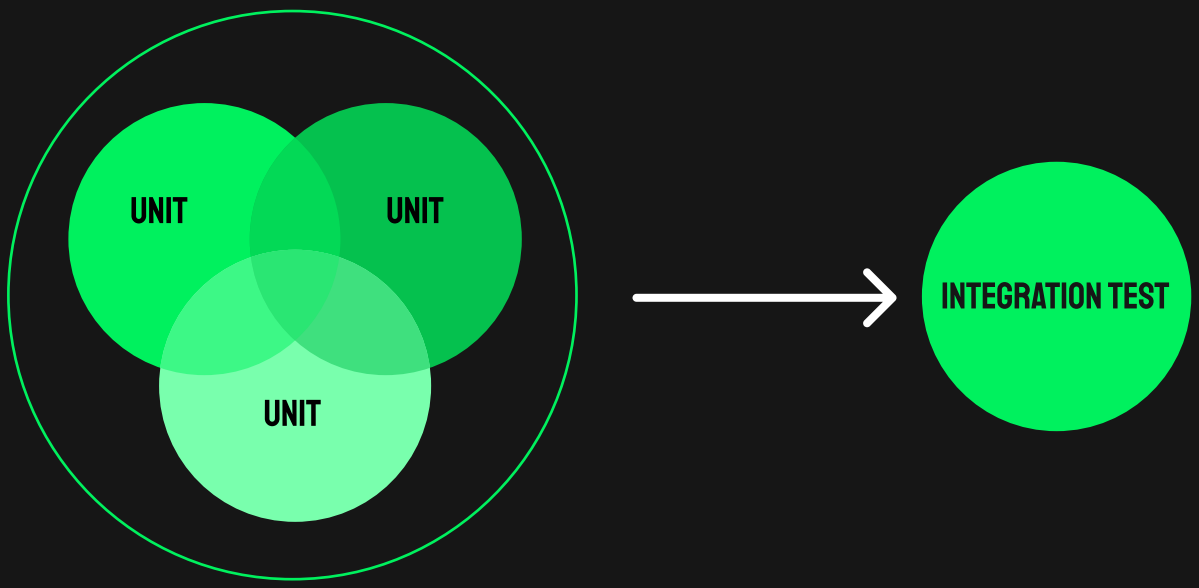
SPEED

MEDIUM

AMOUNT OF COVERED CODE

★★☆

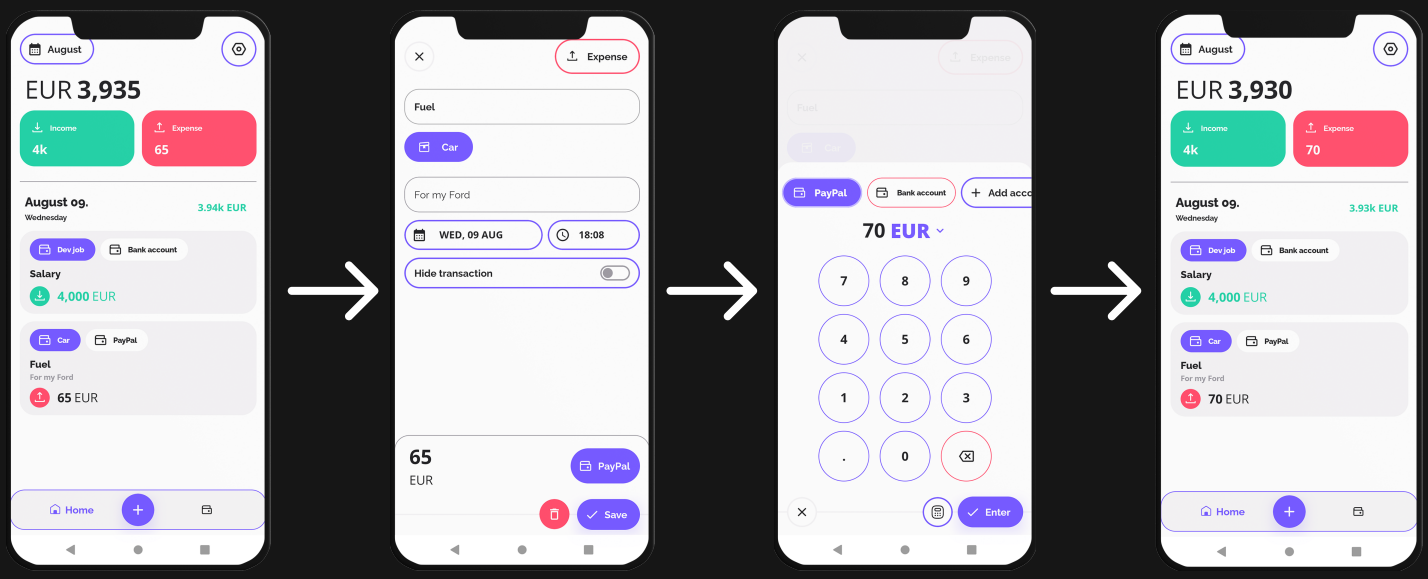
MAINTAINABILITY



END-TO-END TESTS

End-to-end tests test complete user flows in the app.

- That typically involves multiple screens working together
- On Android, these typically involve UI testing



Sample end-to-end test testing to edit a transaction amount in the app.

UI TESTING

UI tests simulate user actions on your app's user interface and assert a visual outcome.

- On Android, we use the Compose testing framework for that (Espresso for XML)

3 TYPES OF UI TESTS

ISOLATED UI TEST

Tests a single UI component in isolation without any interference of other classes (e.g. a ViewModel).

Can also be considered as UI unit tests.

INTEGRATED UI TEST

Tests how a UI component interacts with other classes, such as a ViewModel.

END-TO-END UI TEST

Tests complete user flows, typically across multiple screens.

CAREFUL

UI tests have a high risk of becoming flaky tests. That means they sometimes fail and sometimes pass, providing unreliable results.

You can prevent that by choosing correct view matchers that match unique UI components.

GOOD TO KNOW

Keep Composables light and free of ViewModels. Better just pass your screen's state to your Composables, so you can test them in isolation without needing to pass in a ViewModel instance.

WHAT MAKES A TEST GOOD? - 6 RULES

1. CLEAR AND CONCISE

Keep your test codebase as clean as your real codebase. This allows anyone to understand the test suite which serves as a form of documentation, too.

2. INDEPENDENT

Every test should be completely independent of other tests. A test should never fail because another test changed some form of internal state.

3. REPEATABLE

Good tests shouldn't be flaky. If you run them 100x, they should give you the same result 100x, so you know you can count on it.

4. PRECISE

Each test should have a clear goal and outcome. If it fails, it should give you a strong hint where the bug could lie.

5. FAST

The faster tests are, the more often the team will run them. You can have the best tests - if nobody runs them, they're worthless.

6. COMPREHENSIVE

Good tests test a variety of different scenarios including edge cases. Write tests for common, but also uncommon inputs.

HOW TO KNOW WHAT TO TEST? - 4 QUESTIONS

When thinking about writing a test for a piece of code, ask yourself these 4 questions to get a feeling for whether you should write a test or not. If you struggle to decide, write one.

HOW CRITICAL IS IT FOR THE CORE FUNCTIONALITY?

Write tests for code that is used the most in your app.

WHAT IS THE BUSINESS VALUE?

Write tests for code that creates profit in your app and helps the business to survive (e.g. for buying an in-app subscription)

HOW COMPLEX IS THE CODE?

Write tests for complex code you can't easily look at and tell if it's correct or not.

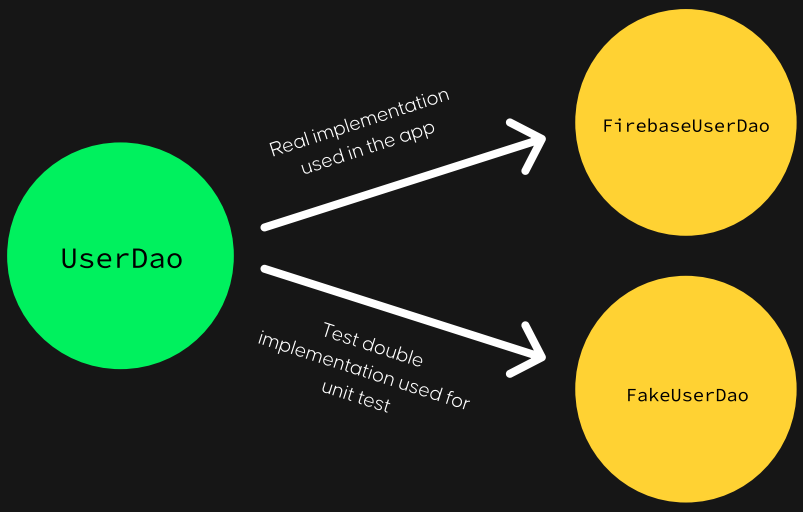
HOW LIKELY IS THE CODE GOING TO CHANGE?

Write tests for code you expect to change in future, so you can make the changes faster and with more confidence.

TEST DOUBLES

A test double is a class used in unit testing when the unit you're testing has dependencies on other parts of the app.

- They allow us to still test the unit in isolation



CAREFUL

Test doubles are meant to replace dependencies you're not actively testing. Therefore, take care that you're not accidentally testing the test double's implementation.

That would result in an always passing test, even if the production code breaks.

THE 5 TYPES OF TEST DOUBLES

1. DUMMY

A dummy is a test double with a completely blank implementation. It can be used if you have to pass an instance of a class to a test, but it's irrelevant for the test.

```
class ShoppingCartCacheDummy: ShoppingCartCache {
    override fun saveCart(items: List<Product>) = Unit

    override fun loadCart(): List<Product> = emptyList()

    override fun clearCart() = Unit
}
```

Example for a dummy

3. FAKE

Fakes are test doubles that simulate the real implementation's behavior in a simplified way. In comparison to stubs, fakes have internal logic.

```
class ShoppingCartCacheFake: ShoppingCartCache {

    private var items = mutableListOf<Product>()

    override fun saveCart(items: List<Product>) {
        this.items.clear()
        this.items.addAll(items)
    }

    override fun loadCart(): List<Product> = items.toList()

    override fun clearCart() {
        items.clear()
    }
}
```

Example for a fake

2. STUB

A stub is comparable to a dummy, but it returns realistic data. Use this if you just need something that provides static data for a test.

```
class ShoppingCartCacheStub: ShoppingCartCache {
    override fun saveCart(items: List<Product>) = Unit

    override fun loadCart(): List<Product> = listOf(
        Product(id = 1, name = "Ice cream", price = 5.0),
        Product(id = 2, name = "Apple", price = 2.0),
        Product(id = 3, name = "Coconut", price = 8.0),
    )

    override fun clearCart() = Unit
}
```

Example for a stub

4. SPY

A spy is a test double that works on the real implementation of the class, but counts how often which function was executed. This can be verified for the test.

```
val shoppingCartCacheSpy = spyk<ShoppingCartCacheImpl>()

verify { shoppingCartCacheSpy.loadCart() }
```

Example for a spy

5. MOCK

A mock is a test double you can freely define the behavior and outputs of for each function. Just like a spy, it counts which method was called how often.

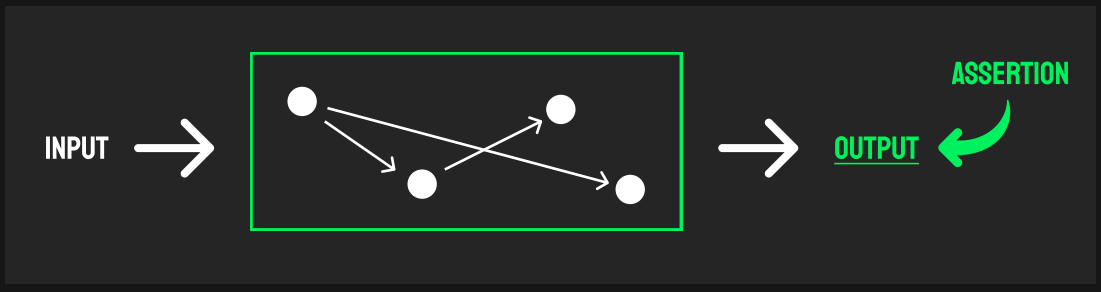
```
val shoppingCartCacheMock = mockk<ShoppingCartCache> {
    every { loadCart() } returns listOf(
        Product(id = 1, name = "Ice cream", price = 5.0)
    )
}

verify { cacheMock.loadCart() }
```

Example for a mock

THE 3 TYPES OF ASSERTIONS

1. OUTPUT OF A FUNCTION

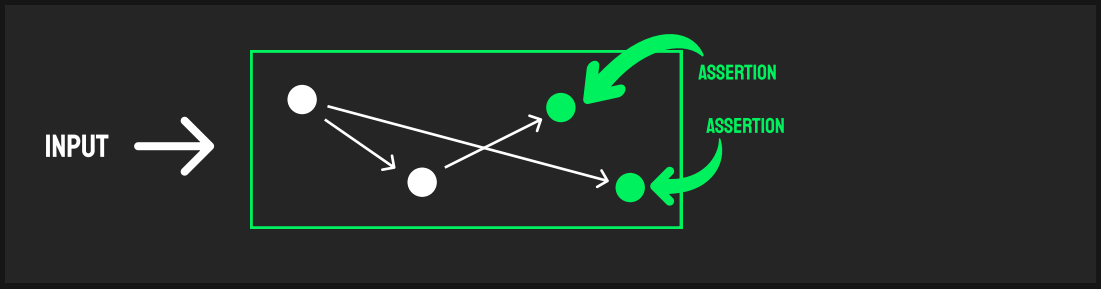
The most simple form of assertion. A function with a given input is executed and you assert if the return value is what is expected.



2. STATE

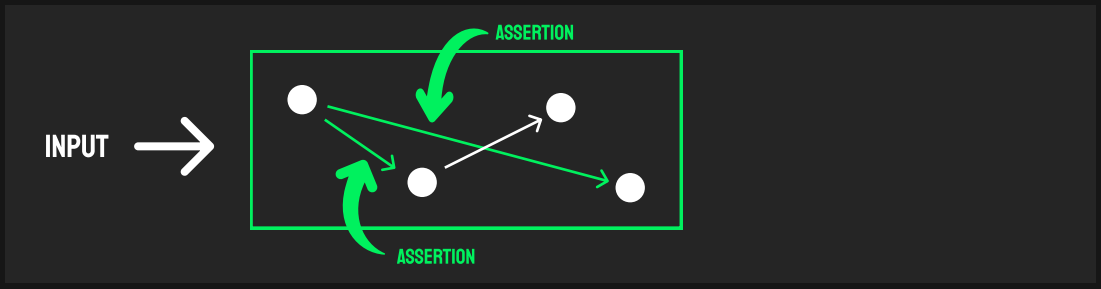
A function might not directly return a specific output, but rather cause a change of state.

This type of assertion runs on the state itself rather than on a direct output.



3. COMMUNICATION

You might also want to test that a specific call to a function was executed. Those assertions are called communication assertions and typically involve mocks.



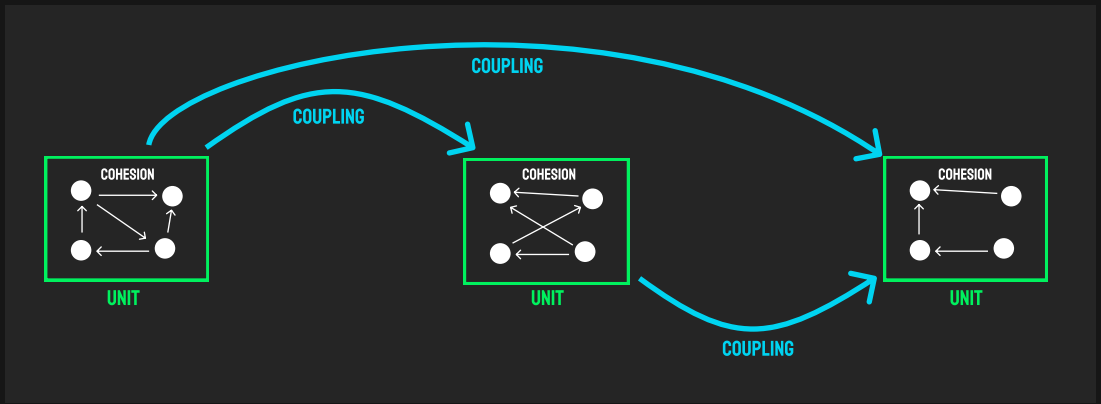
WHAT MAKES CODE TESTABLE?

1. HIGH COHESION

Cohesion refers to how closely the responsibilities of a unit are related to each other. In other words: How much does a class use what it directly provides?

2. LOW COUPLING

Coupling refers to how much a unit interacts with other parts of the system. This should be as low as possible.

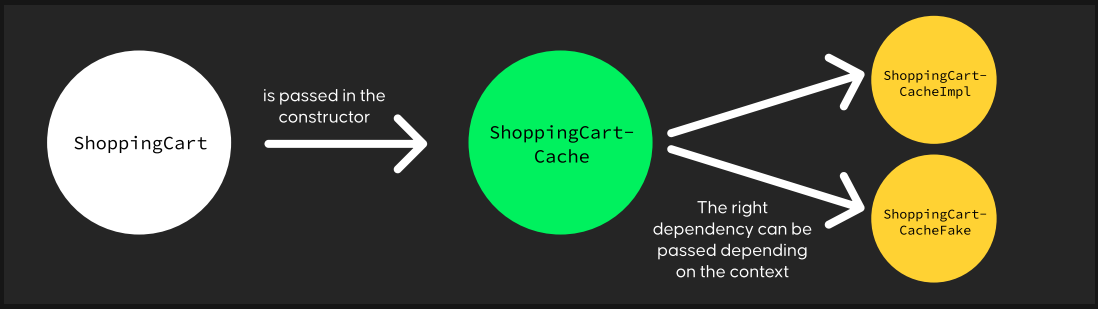


3. PROPER ABSTRACTIONS

If a class A is coupled to another class B (which is unavoidable), create a proper abstraction for class B, in order to test class A in isolation with a test double.

4. DEPENDENCY INJECTION

Dependency injection allows you to pass test doubles for an instance of a class under test. Avoid initializing abstractions as private class fields and rather pass them in the constructor.



MUTATION TESTING

Mutation tests are used to test your tests.

- After writing a passing test, you intentionally break the code under test
- The test is then expected to fail as well
- If it doesn't, you know there is an issue in the test

GOOD TO KNOW

The more complex your tests get, the more likely it is you accidentally made a mistake in the test. That's why it's a good idea to make mutation testing a habit after every set of tests you write.