Mobile Applications

8. Automated tests in Android

Boni García

boni.garcia@uc3m.es

Telematic Engineering Department
School of Engineering

2024/2025

uc3m | Universidad Carlos III de Madrid

Table of contents

1. Introduction
2. Software testing
3. Test automation tools
4. Automated tests in Android
5. Continuous integration
6. Takeaways

1. Introduction

- In this course, we have studied the basics of Android development
- To verify that the app being developed work as expected, we deploy the app in an Android Virtual Device (AVD) or a physical device and interact with the app manually
- This is a type of manual testing, and it is done always by developers as part of its daily job
- In addition to manual testing, in real software development, it is very important to carry out **automated testing** as well

Table of contents

1. Introduction
2. Software testing
   - Test automation
   - Levels of testing
   - Types of testing
3. Test automation tools
4. Automated tests in Android
5. Continuous integration
6. Takeaways

2. Software testing

- **Software testing** (or simply testing) consists of the dynamic evaluation of a piece of software, called System Under Test (SUT), to ensure it is working as intended and meets user expectations
- We distinguish two big categories of software testing:

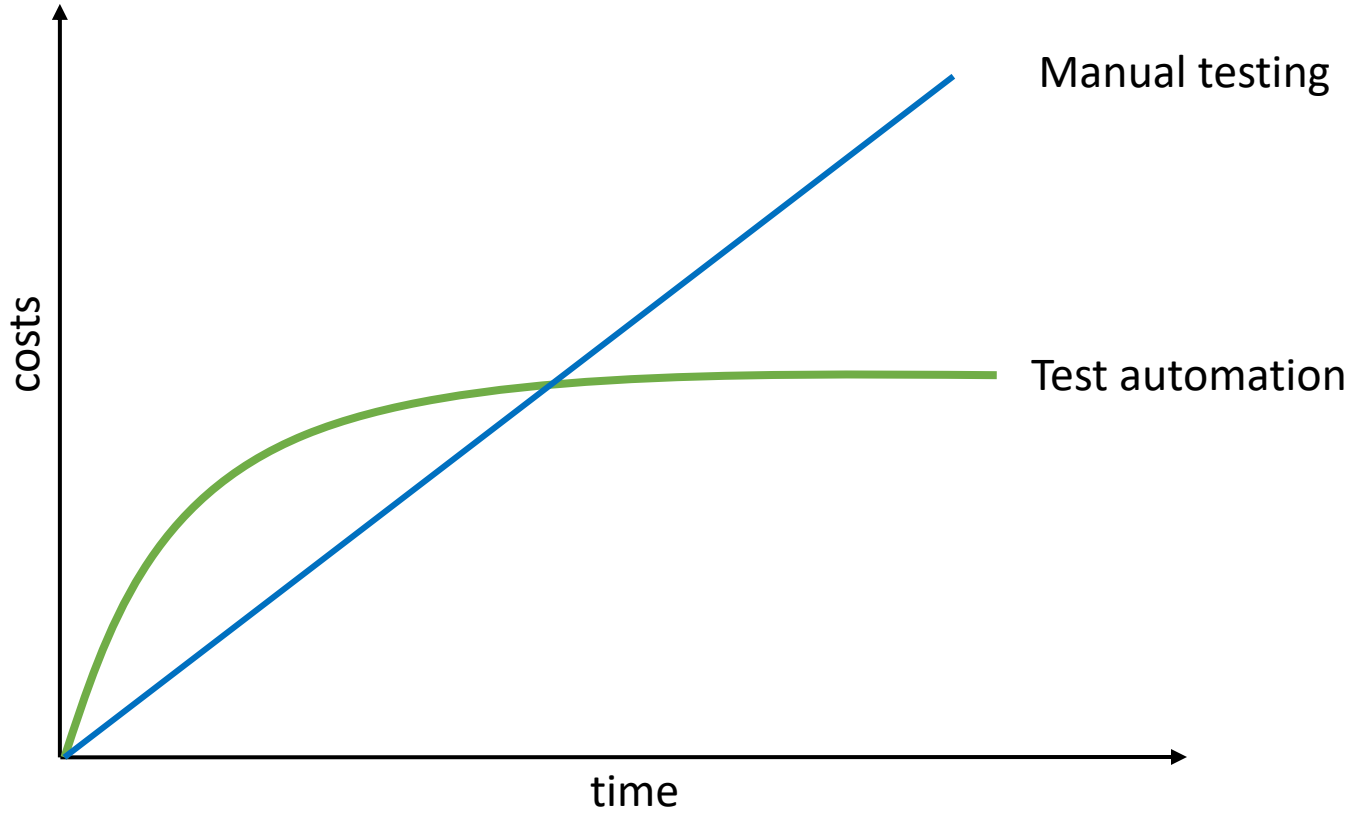
1. **Manual testing**, a person (typically a developer, tester, or even the final user) evaluates the SUT
2. **Automated testing**, a software engineer (such as a developer or tester) implements a piece of code (called **test case** or simply **test**) and use specific software **tools** to control their execution against the SUT

2. Software testing - Test automation

- The main benefits of test automation are the following:
  - Early detection of software defects (usually called *bugs*) in the SUT
  - Faster feedback: automated tests run faster than manual tests, enabling quicker feedback on code changes (to avoid regressions)
    - A regression is a bug that appears in a previously working part of the software after changes have been made
  - Repeatability: automated tests allows us to create complex systems by ensuring consistent test execution and results
  - Scalability: automated tests can be scaled across multiple platforms (browsers, devices, operating systems, etc.)
  - Efficiency: higher initial setup cost but reduces long-term expenses by minimizing repetitive manual effort

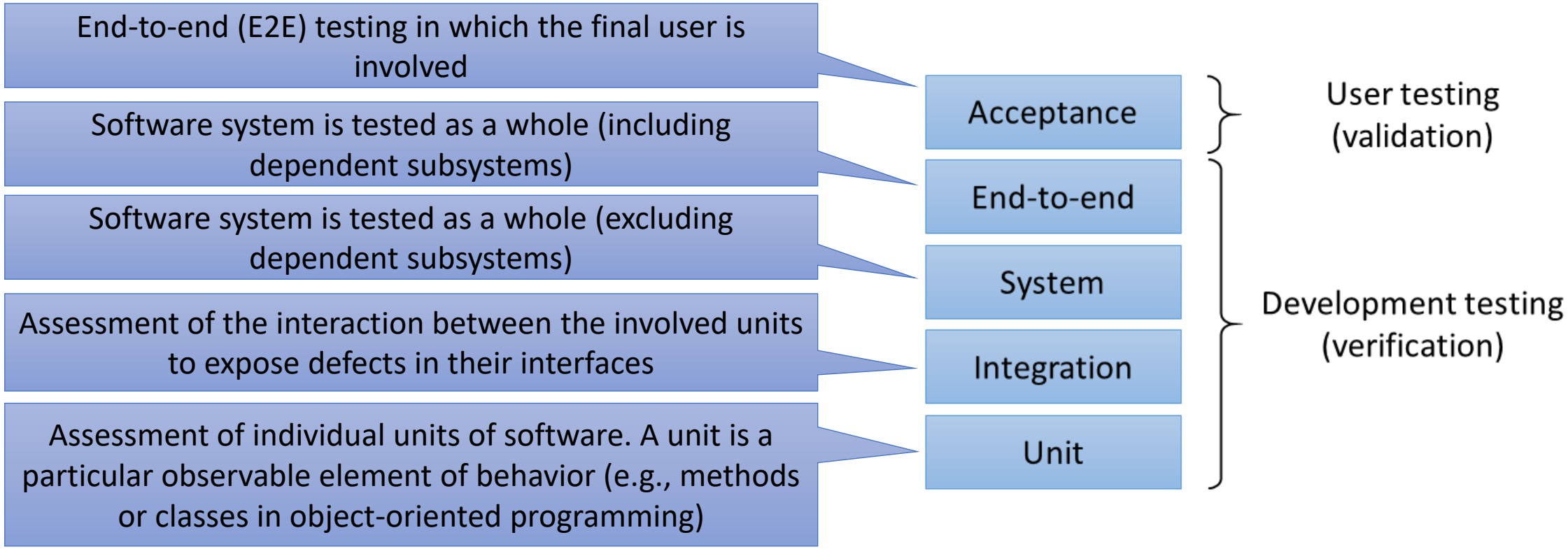
2. Software testing - Test automation

Compared to manual testing, test automation has a high initial investment but it allows to save cumulative costs in the long-term

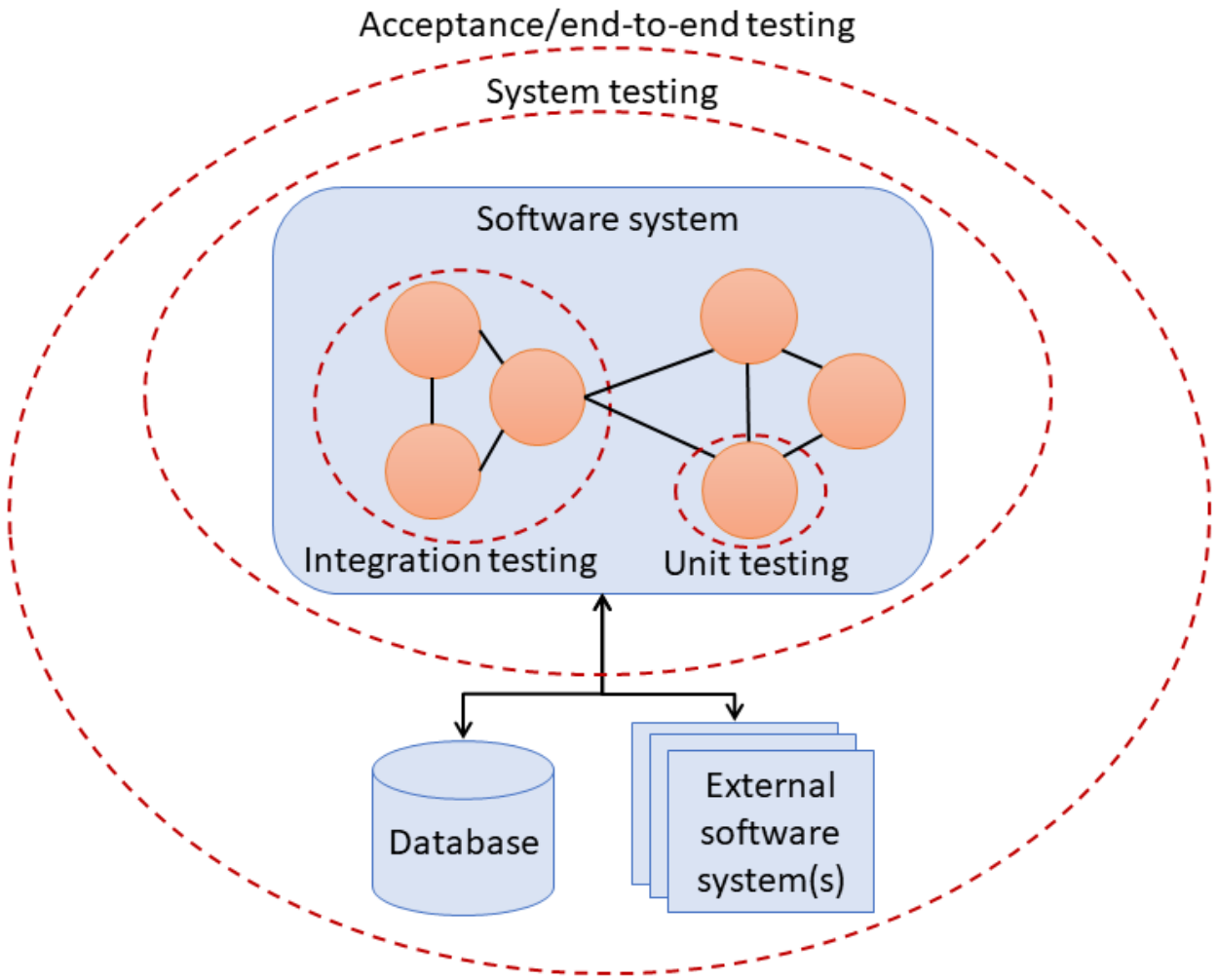


2. Software testing - Levels of testing

- Depending on the size of the SUT, we can define different **levels of testing**
  - These levels define several categories in which software teams divide their testing efforts



2. Software testing - Levels of testing



2. Software testing - Types of testing

- Depending on the strategy for designing test cases, we can implement different types of tests:
  - **Functional testing** (also known as behavioral or closed-box testing). Evaluates the compliance of a piece of software with the expected behavior (i.e., its functional requirements)
  - **Structural testing** (also known as white-box testing). Determines if the program-code structure is faulty. To that aim, testers should know the internal logic of a piece of software
  - **Nonfunctional testing**, includes testing strategies that assess the quality attributes of a software system (i.e., its nonfunctional requirements)
    - Such as performance, security, usability, or accessibility testing, among others

The difference between these testing types is that functional tests are responsibility-based, while structural tests are implementation-based

Table of contents

1. Introduction
2. Software testing
3. Test automation tools
   - Unit tests
   - E2E tests for mobile apps
4. Automated tests in Android
5. Continuous integration
6. Takeaways

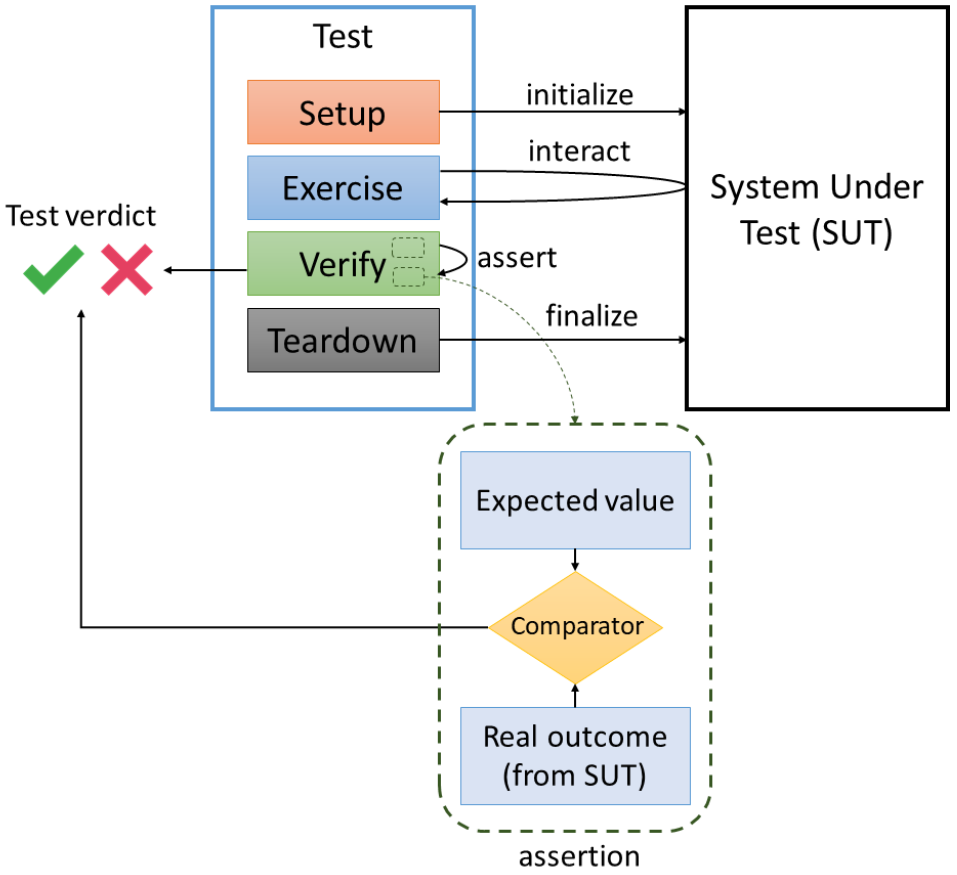
3. Test automation tools

- Automated testing requires the use of some tooling to implement, execute, and control the automated tests effectively
- One of the most relevant categories for testing tools is the **unit testing frameworks**
- A unit testing framework allows developers to write and run repeatable tests
  - A library is a collection of code that developers can call using an API to solve a given problem
  - A framework is collection of libraries, tools, and best practices that provides a structure for developing software
    - Therefore, a framework is typically more complex and restricted than a library since it defines a skeleton where the piece of software using it implements its logic

3. Test automation tools - Unit tests

- The following picture illustrates the typical steps of a unit test

The steps “setup” and “teardown” are optional



3. Test automation tools - Unit tests

- In Java, some of the most relevant unit testing frameworks are:
  - JUnit (version 4 or 5)
  - TestNG

https://junit.org/junit4/

https://junit.org/junit5/

https://testng.org/

Android Studio uses JUnit 4 by default

3. Test automation tools - E2E tests for mobile apps

- Automated testing is an essential process in the development of production-ready apps
- We need specific tooling to test automatically mobile apps from its UI

https://developer.android.com/training/testing/espresso

https://appium.io/

In this course, we focus in the specific testing libraries for Jetpack Compose apps

https://developer.android.com/develop/ui/compose/testing

4. Automated tests in Android - Compose testing

- Compose provides a set of testing APIs to find elements, verify their attributes, and perform user actions
- The Compose testing cheat sheet provides a quick reference of some of the most useful Compose test APIs

https://developer.android.com/develop/ui/compose/testing/testing-cheatsheet

Table of contents

1. Introduction
2. Software testing
3. Test automation tools
4. Automated tests in Android
   - Android tests
   - Unit tests
5. Continuous integration
6. Takeaways

4. Automated tests in Android

- Android Studio provides seamless integration with different tools to carry out automated testing in a seamless manner
  - Unit testing: JUnit
  - E2E testing: Jetpack Compose testing libraries
- The required testing dependencies (included by default in any new Android Studio project) are:

build.gradle.kts (app)

```
testImplementation(libs.junit)
androidTestImplementation(platform(libs.androidx.compose.bom))
androidTestImplementation(libs.androidx.ui.test.junit4)
```

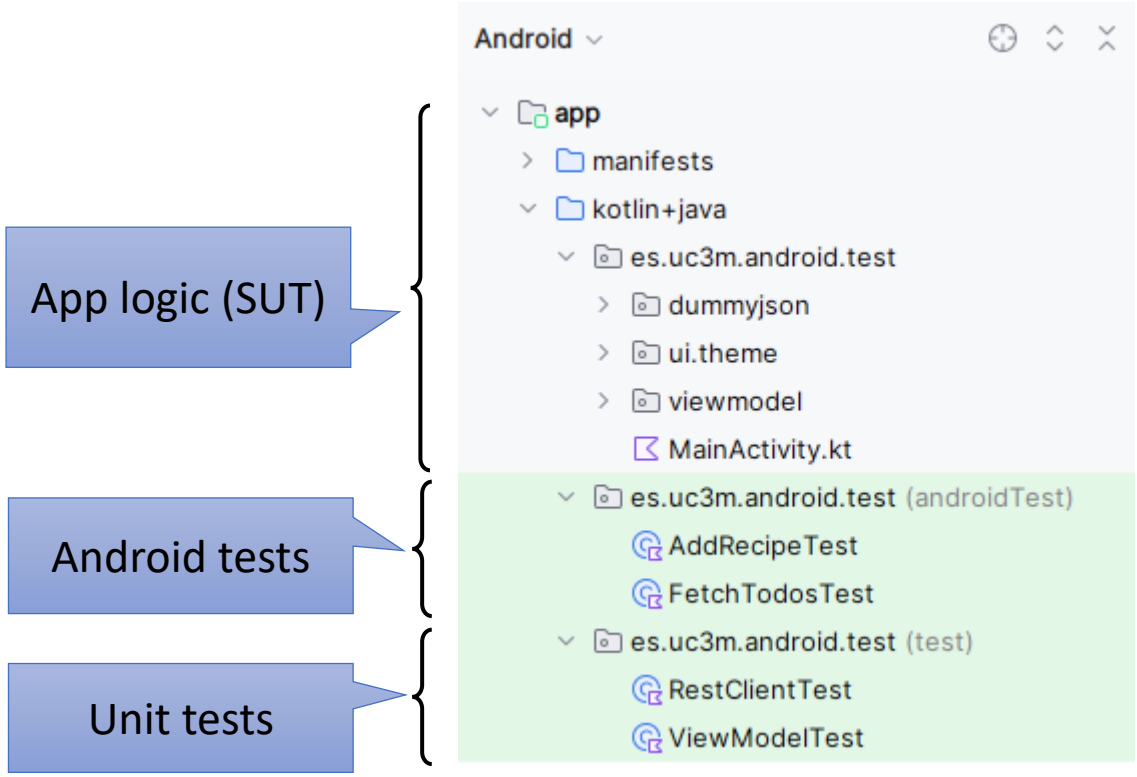
libs.version.toml

```
[versions]
junit = "4.13.2"
composeBom = "2025.04.00"

[libraries]
junit = { group = "junit", name = "junit", version.ref = "junit" }
androidx-ui-test-manifest = { group = "androidx.compose.ui", name = "ui-test-manifest" }
androidx-ui-test-junit4 = { group = "androidx.compose.ui", name = "ui-test-junit4" }
```

https://developer.android.com/studio/test
https://developer.android.com/training/testing
https://developer.android.com/develop/ui/compose/testing

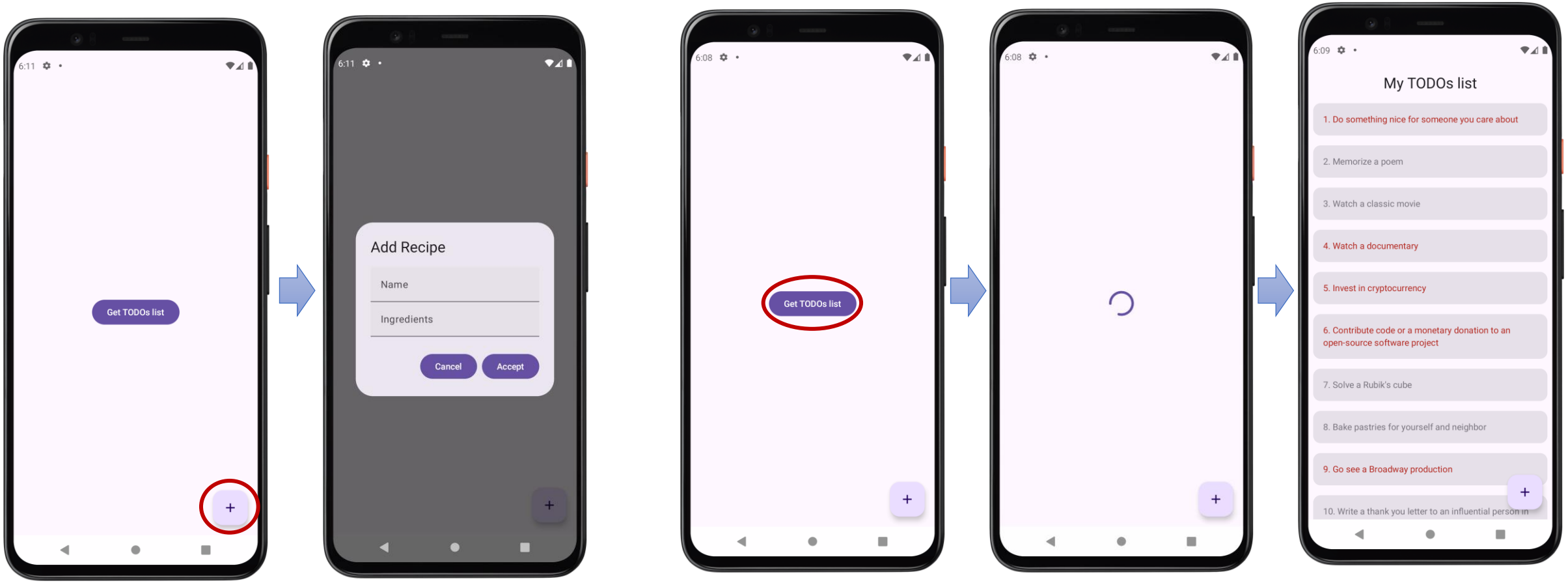
4. Automated tests in Android

- Android makes the following distinction between tests:
  - **Units tests** (sometimes called *local* tests): Small and fast, isolating the subject under test from the rest of the app
  - **Android tests**: run on an Android device, either physical or emulated. There are two types:
    - UI tests (E2E tests): These tests launch the app and then interacting with it through its UI
    - Context tests (integration tests): This type of tests uses the context object to evaluate some behavior of the SUT



4. Automated tests in Android - Android tests

- In the following examples, we use the demo app implementing a REST client as SUT:



4. Automated tests in Android - Android tests


```
class AddRecipeTest {

    private val context = InstrumentationRegistry.getInstrumentation().targetContext

    @get:Rule
    val composeTestRule = createAndroidComposeRule<MainActivity>()

    @Test
    fun addRecipeTest() {
        // Exercise: click on +
        val add = context.getString(R.string.add)
        composeTestRule.onNodeWithContentDescription(add).performClick()

        // Exercise: add recipe
        composeTestRule.onNode(hasText(context.getString(R.string.name)))
            .performTextInput("My recipe")
        composeTestRule.onNode(hasText(context.getString(R.string.ingredients)))
            .performTextInput("My ingredients")
        val accept = context.getString(R.string.accept)
        composeTestRule.onNodeWithText(accept).performClick()

        // Verify: we're back to home
        composeTestRule.onNodeWithContentDescription(accept).assertIsNotDisplayed()
        composeTestRule.onNodeWithContentDescription(add).assertIsDisplayed()
    }

}
```

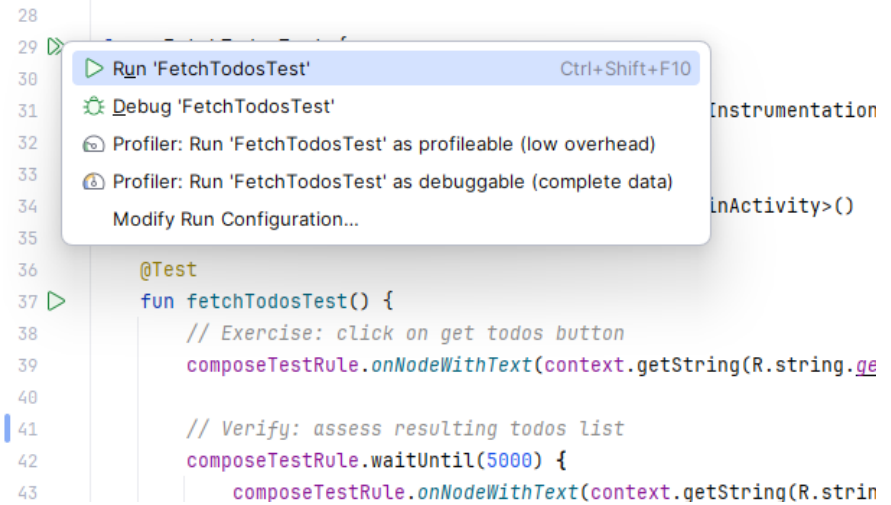
This test uses our SUT's UI in the same way that a final user would (exercise) and the verifies the UI is as expected

We can execute the test in Android Studio using this button

4. Automated tests in Android - Android tests


```
class FetchTodosTest {

    private val context = InstrumentationRegistry.getInstrumentation().targetContext

    @get:Rule
    val composeTestRule = createAndroidComposeRule<MainActivity>()

    @Test
    fun fetchTodosTest() {
        // Exercise: click on get todos button
        composeTestRule.onNodeWithText(context.getString(R.string.get_todos)).performClick()

        // Exercise: verify resulting todos list
        composeTestRule.waitUntil(5000) {
            composeTestRule.onNodeWithText(context.getString(R.string.my_todos)).isDisplayed()
        }
        composeTestRule.onNodeWithTag("todos").onChildren().fetchSemanticsNodes().isNotEmpty()
    }

}
```

This test verifies other part of the UI, waiting until the results are available in the screen as expected



4. Automated tests in Android - Unit tests

- Unit testing in Android apps can be challenging since we need to isolated the unit under test
- A potential “unit” can be those classes used to implement REST clients

```
interface DummyJsonService {

    @GET("todos")
    suspend fun getTodos(): Response<Todos>

    @POST("recipes/add")
    suspend fun addRecipes(@Body recipe: Recipe): Response<Recipe>

}
```

The problem is that these clients are implemented with coroutines

build.gradle.kts (app)

```
testImplementation(libs.kotlinx.coroutines.test)
```

libs.version.toml

```
[versions]
kotlinxCoroutinesTest = "1.9.0"

[libraries]
kotlinx-coroutines-test = { module = "org.jetbrains.kotlinx:kotlinx-coroutines-test", version.ref = "kotlinxCoroutinesTest" }
```

The solution is to use an specific library for testing coroutines

4. Automated tests in Android - Unit tests

```
class RestClientTest {

    @Test
    fun dummyJsonTest() = runTest {
        // Exercise
        val response = DummyJsonClient.apiService.getTodos()

        // Verify
        assertTrue(response.isSuccessful)
        var todos = response.body()?.todos!!
        println(">>> todos: $todos")
        assertTrue(todos.isNotEmpty())
    }
}
```

`@Test` annotation marks this as a test function that will be executed by the testing framework (i.e., JUnit 4)

`runTest` is a coroutine test runner that handles asynchronous code execution

4. Automated tests in Android - Unit tests

- Another potential “unit” can be those classes used to implement view models

```
class RestViewModel : ViewModel() {
    private val _todos = MutableStateFlow<List<Todo>>(emptyList())
    val todos: StateFlow<List<Todo>> get() = _todos

    fun fetchTodos() {
        viewModelScope.launch {
            _isLoading.value = true
            try {
                val response = DummyJsonClient.apiService.getTodos()
                if (response.isSuccessful) {
                    _todos.value = response.body()?.todos!!
                }
            } catch (e: Exception) {
                _toastMessage.value = e.message
            } finally {
                _isLoading.value = false
            }
        }
    }

}
```

The problem is that these view models uses a coroutine scope provided by Android (*viewModelScope*). But we want to execute a unit test, not and Android test

4. Automated tests in Android - Unit tests

```
@OptIn(ExperimentalCoroutinesApi::class)
class ViewModelTest {
    private val testDispatcher = UnconfinedTestDispatcher()

    @Before
    fun setup() {
        Dispatchers.setMain(testDispatcher)
    }

    @Test
    fun viewModelTest() = runTest {
        val viewModel = RestViewModel()
        viewModel.fetchTodos()

        val await = Awaitility.await().atMost(Duration.ofSeconds(5))
        await.until { viewModel.todos.value.isNotEmpty() }

        val todos = viewModel.todos.value
        println("*** todos: $todos")
    }

    @After
    fun tearDown() {
        Dispatchers.resetMain()
    }

}
```

A solution is to use a coroutine test dispatcher (i.e., a thread to be used instead the thread pool provided in the coroutine Android scope)

But in this case, in addition to the test runner (*runTest*), we need some mechanism to wait the response, such as Awaitility

build.gradle.kts (app)

```
testImplementation(libs.awaitility)
```

libs.version.toml

```
[versions]
awaitility = "4.3.0"

[libraries]
waitility = { module = "org.awaitility:awaitility", version.ref = "awaitility" }
```

Table of contents

1. Introduction
2. Software testing
3. Test automation tools
4. Automated tests in Android
5. Continuous integration
   - Build server
   - GitHub Actions
6. Takeaways

5. Continuous integration

- **Continuous Integration** (CI) is a software development strategy where members of a software project build, test, and integrate their work continuously in three separate stages:

Source code is typically managed using a **version control system** (like Git or CVS)

A **build server**, also known as a CI server, is a dedicated system that automates the process of building, testing, and deploying software applications

5. Continuous integration - Build server

- A **build server** is server-side infrastructure that implement CI pipelines (sometimes called *workflows*)
  - A CI pipeline is a series of steps executed to build/test/deploy a given software
- Some popular build servers are:

GitHub Actions

https://docs.github.com/en/actions

CI/CD

https://docs.gitlab.com/ee/ci/

Jenkins

https://www.jenkins.io/

Bamboo

https://www.atlassian.com/software/bamboo

5. Continuous integration - GitHub Actions

- You can find a separate GitHub repository (different than the usual for examples) with a complete example of app, tests, and CI configuration
- Each time a new commit is done in the repo, the whole test suite (local and instrumented) is executed in GitHub Actions
  - When some test fails (*regression*), the development team is notified

https://github.com/bonigarcia/android-basic-app

Table of contents

1. Introduction
2. Software testing
3. Test automation tools
4. Automated tests in Android
5. Continuous integration
6. Takeaways

6. Takeaways

- Software testing consists of the dynamic evaluation of a piece of software (SUT), giving a verdict about it
- In automated testing, we use specific software tools to develop test scripts and control their execution against the SUT
- Android Studio provides seamless integration with JUnit 4 and Jetpack Compose for unit and Android tests
- Development teams usually a server-side infrastructure called a build server (such as GitHub Actions) to implement CI pipelines and execute a suite of automated tests during the development lifecycle