Testing (Practice)

Week 7-2

Any fool can write code that a computer can understand.
Good programmers write code that humans can understand.

— Martin Fowler

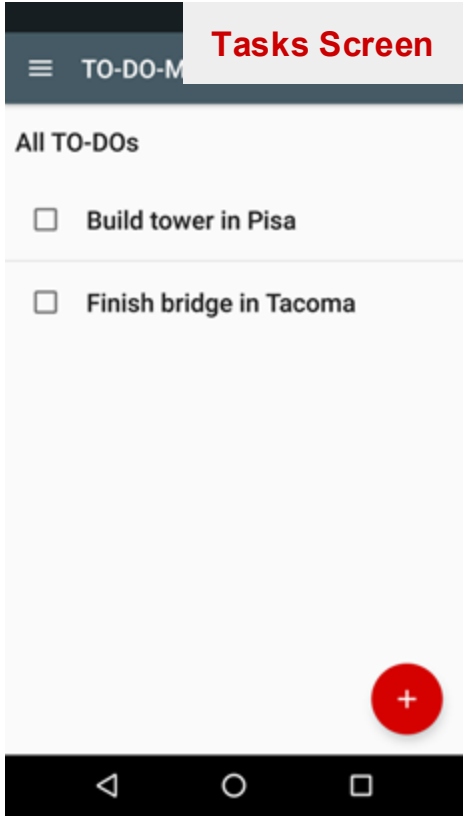
Objectives

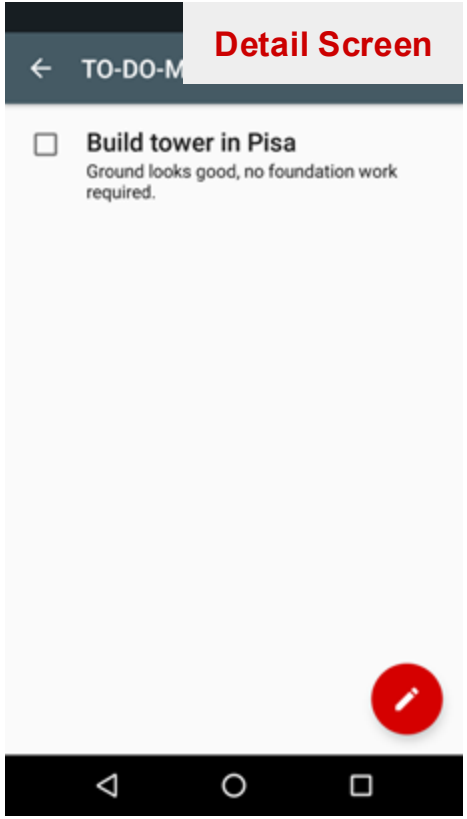
- App Testing Practice using Todo app 😉
- Unit Tests in Two Layers:
  - **Model** Testing (Junit)
  - **ViewModel** Testing (Mockito + Hilt) ***- today's main goal!***

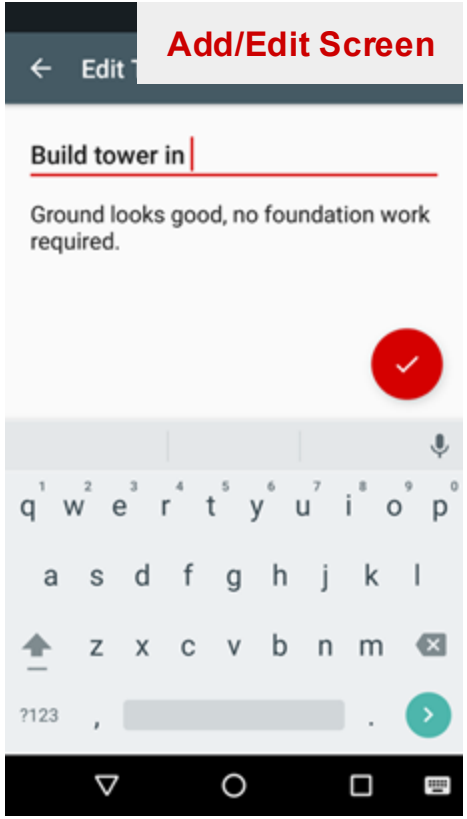
Recap: App Specification

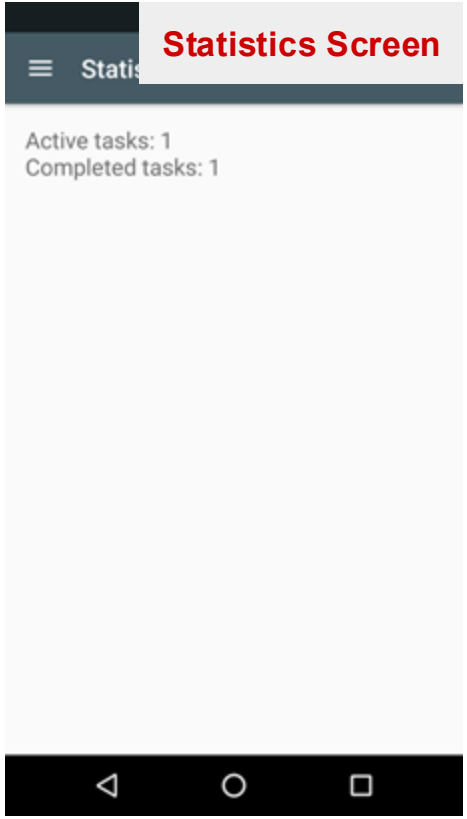
App skeleton

App specification









Recap: Get the Skeleton Code

- Use Git
  - Clone the skeleton repository in the previous slide
- Or you could download the raw zip file

- **Submission Preparation**
  - Create a `submission` directory
  - Make subdirectories **`exercise01 - exercise04`**
  - You have to submit these files at the end of this week!

Recap: Testing Strategy

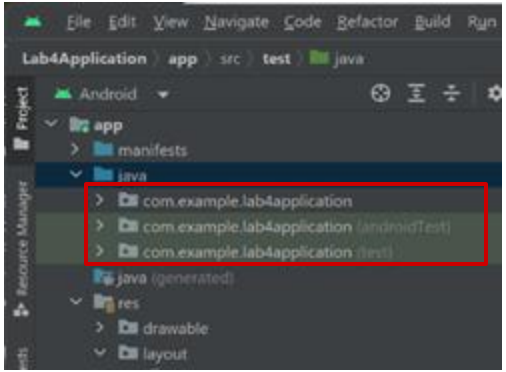
This week's scope!

1. First you'll **unit test** the model.
2. Then you'll use a **test double** in the view model, which is necessary for unit testing and integration testing the view model.
3. Next, you'll learn to write **integration tests** for fragments and their view models.
4. Finally, you'll learn to write **integration tests** that include the Navigation component.

https://developer.android.com/codelabs/advanced-android-kotlin-training-testing-test-doubles#2

Android Studio: Testing Interface

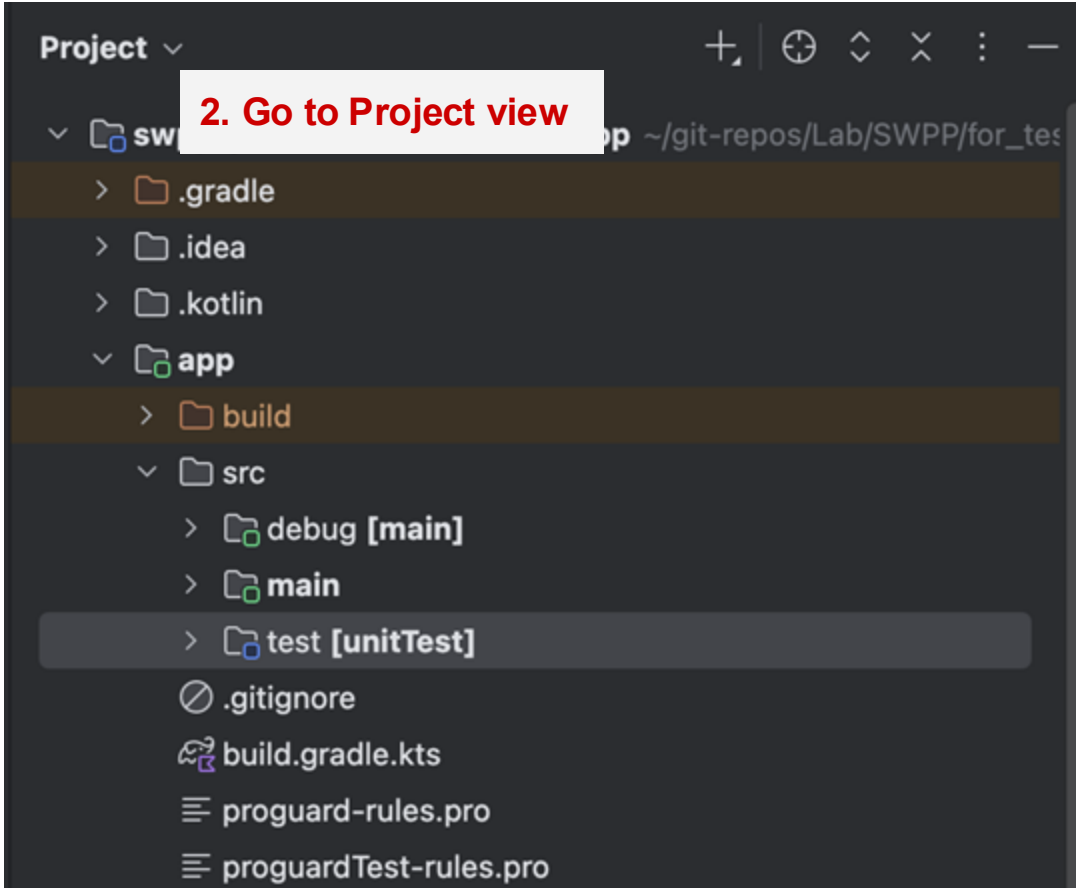
Test Source Sets



3 **source sets**:

- `main`: contains app code
- `androidTest`: contains instrumented tests
  - Runs on real/emulated device
  - Slow
  - High fidelity
  - For integration, E2E, UI tests
- **`test`**: contains local tests
  - Runs on JVM (not on real/emulated device)
  - Fast
  - Low fidelity
  - For unit tests

Tip: Finding the Test Directories (1/2)

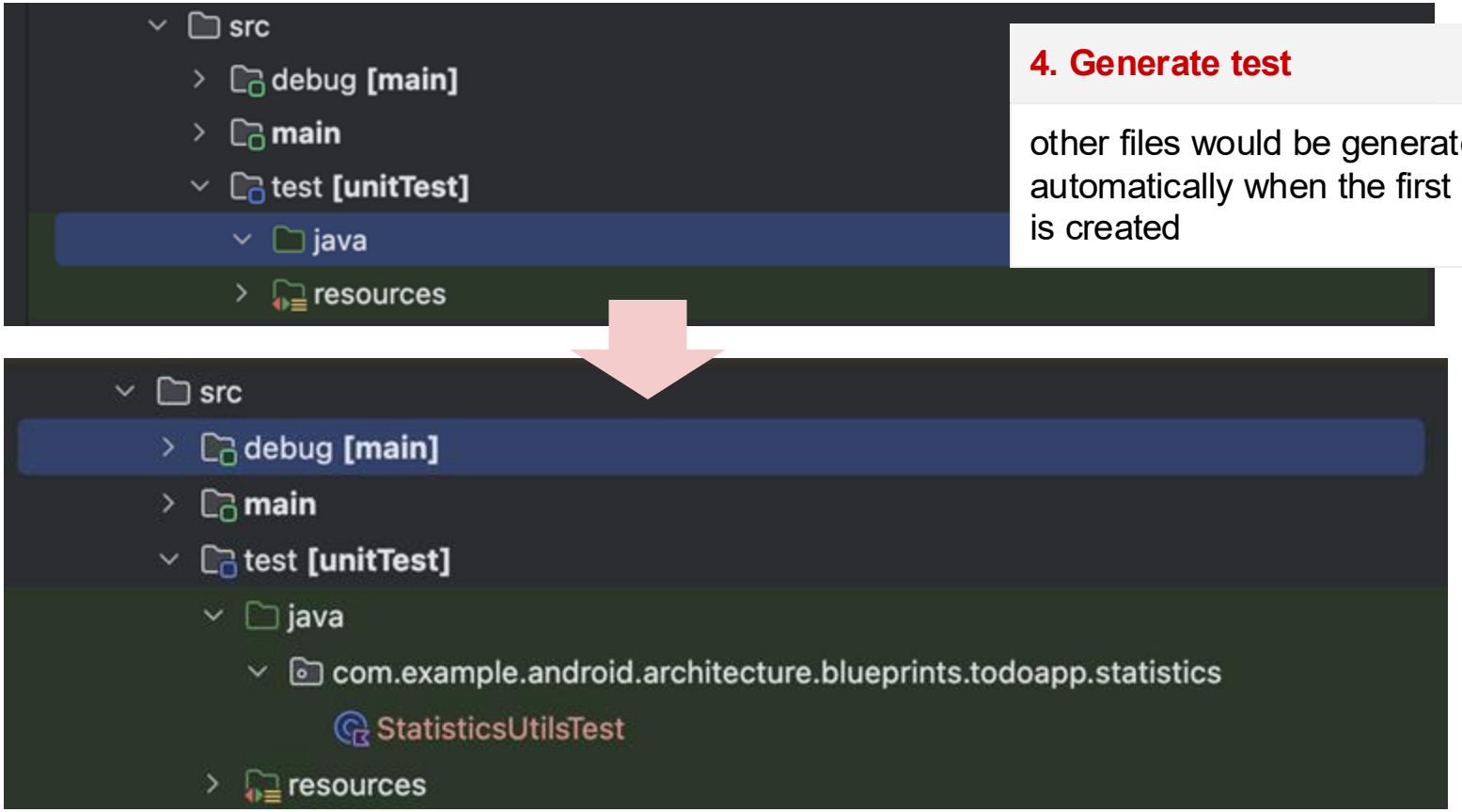


1. First build!

3. Check `app/src/test`

If you don't have one, create **`app/src/test/java/`** manually

Tip: Finding the Test Directories (2/2)

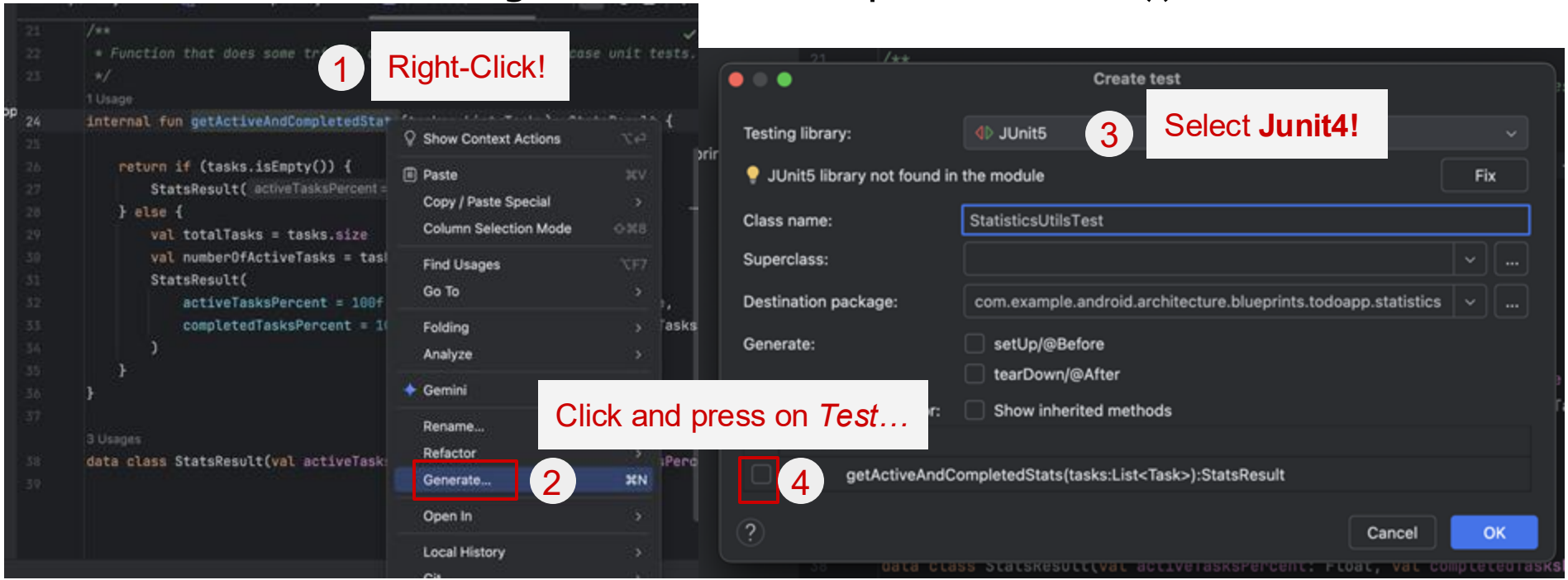


4. Generate test

other files would be generated automatically when the first test file is created

Recap: Writing Unit Tests

- We wrote a unit test for `StatisticUtils.getActiveAndCompletedStats()` last time



Recap: JUnit4

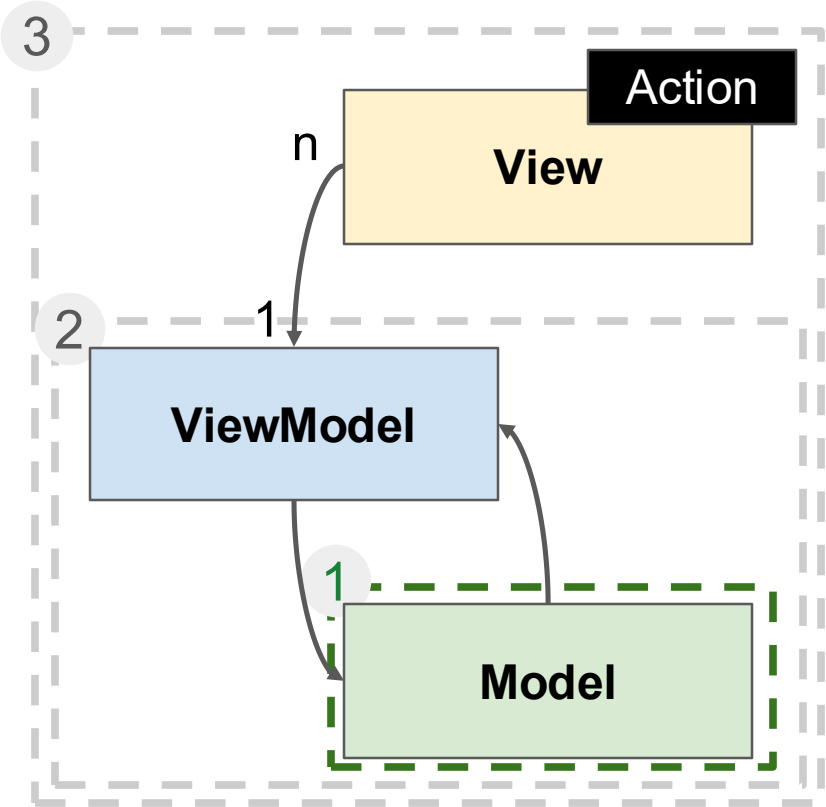
- Unit testing framework for Java and Kotlin (family of xUnit)
- Used to write and run repeatable automated unit tests
- Requires Java 6 (or higher)

JUnit annotation

JUnit assertion

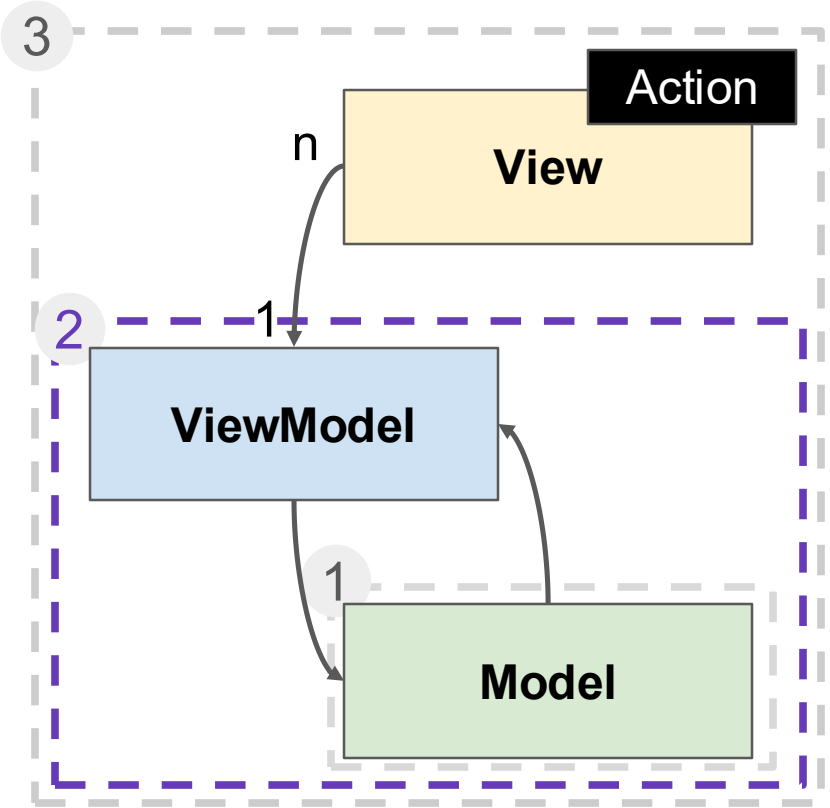
```
@Test
fun isPrime_two_returnsTrue() {
    // GIVEN input 2
    val input = 2

    // WHEN isPrime is called
    val result = isPrime(input)

    // THEN return true
    assertTrue(result)
}
```

Now We Know How to Test a Model!



Mockito Basics

How to Test a ViewModel



: Lots of data involved…

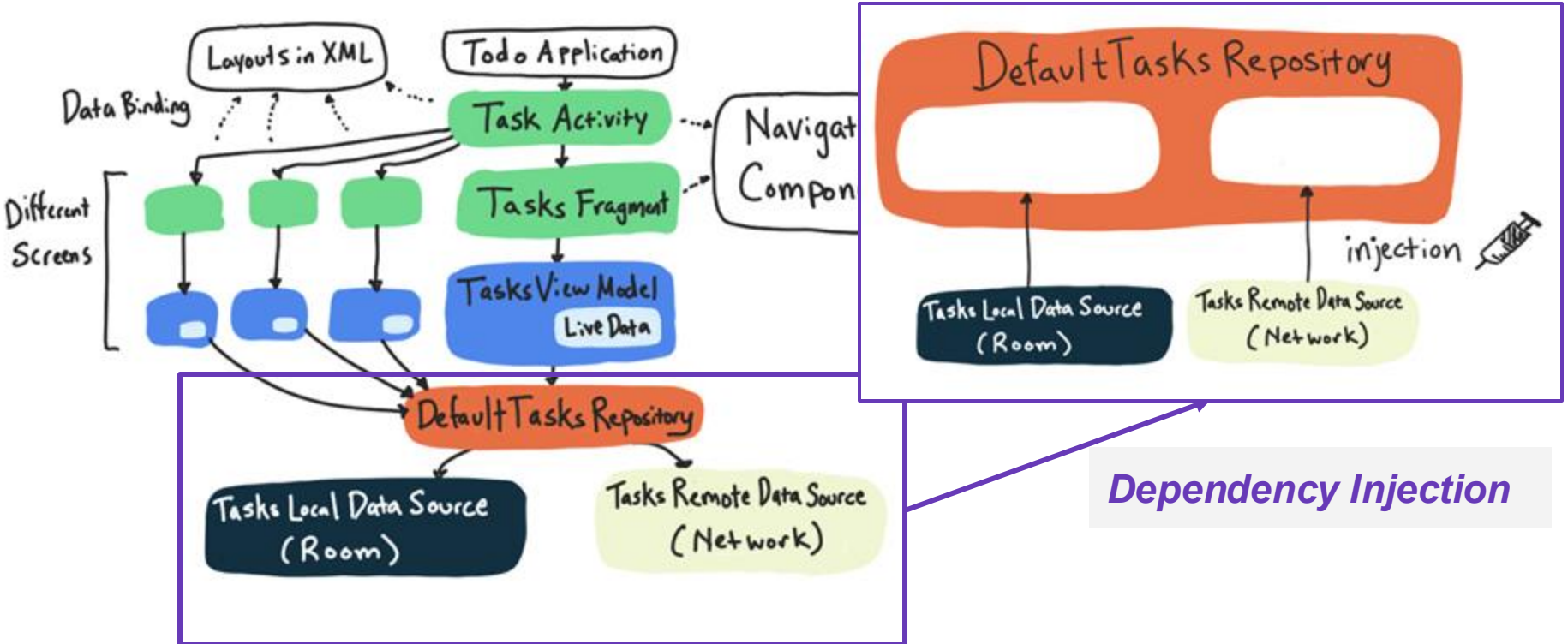
Challenges in Testing a ViewModel

```
override suspend fun createTask(title: String, description: String): String {
    // ID creation might be a complex operation so it's executed using the supplied
    // coroutine dispatcher
    val taskId = withContext( context = dispatcher) {
        UUID.randomUUID().toString()
    }
    val task = Task(
        title = title,
        description = description,
        id = taskId,
    )
    localDataSource.upsert( task = task.toLocal())
    saveTasksToNetwork()
    return taskId
}
```

Hard to implement everything in tests :(

Sol: Testing in an Isolated Environment

- Essential when testing ViewModel



Dependency Injection

Mockito

- Mocking framework for Java (Kotlin)
- Allows convenient creation of **substitutes of real objects** for testing purposes
- Used for test doubles in unit testing
- We'll use Mockito **5.11.0**

Mockito Setup

- Add mockito dependency to your `build.gradle (app)`

```
testImplementation "org.mockito:mockito-core:5.11.0"
```

& Sync

Dependencies applied to ALL source sets

Dependencies applied to the test source set

Dependencies applied to the androidTest source set

```
dependencies {
    def lifecycle_version : String = "2.6.2"
    implementation "androidx.lifecycle:lifecycle-viewmodel:$lifecycle_version"
    implementation "androidx.lifecycle:lifecycle-livedata:$lifecycle_version"
    implementation 'androidx.appcompat:appcompat:1.6.1'
    implementation 'com.google.android.material:material:1.9.0'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.4'

    testImplementation 'junit:junit:4.13.2'
    testImplementation "org.mockito:mockito-core: 5.11.0

    androidTestImplementation 'androidx.test.ext:junit:1.1.5'
    androidTestImplementation "org.mockito:mockito-core:5.5.0"
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.5.1'
```

Mockito Setup

- Add mockito dependency to your `build.gradle (app)`

```
testImplementation "org.mockito:mockito-core:5.11.0"
```

& Sync

```
57    testImplementation(libs.kotlinx.coroutines.test)
58    testImplementation(libs.androidx.navigation.testing)
59    testImplementation(libs.androidx.test.espresso.core)
60    testImplementation(libs.androidx.test.espresso.contrib)
61    testImplementation(libs.androidx.test.espresso.intents)
62    testImplementation(libs.google.truth)
63    testImplementation(libs.androidx.compose.ui.test.junit)
64    testImplementation("org.mockito:mockito-core:5.11.0")
65
66    // JVM tests - Hilt
67    testImplementation(libs.hilt.android.testing)
```

Unit Testing TasksViewModel

- Let's try to write unit test for `TasksViewModel.completeTask()`
- We want:
  - GIVEN a use case - 3 tasks, with one active and two completed
  - WHEN `completeTask` for the remaining active task is called
  - THEN change completion state & show Snackbar message

```kotlin
fun completeTask(task: Task, completed: Boolean) = viewModelScope.launch{
    if (completed) {
        taskRepository.completeTask( taskId = task.id)
        showSnackbarMessage("Task marked complete")
    } else {
        taskRepository.activateTask( taskId = task.id)
        showSnackbarMessage("Task marked active")
    }
}
```

```kotlin
6 Usages
private fun showSnackbarMessage(message: Int) {
    _userMessage.value = message
}
```

Unit Testing TasksViewModel: Problem 1

- What happens if `TasksViewModel.completeTask()` has a bug and doesn’t show the expected messages? What if `showSnackbarMessage()` is faulty?
- We're not testing `TasksViewModel.completeTask()` in an *isolated environment*

Unit Testing TasksViewModel: Problem 2

- Requires Android framework dependency (context)

```
@Test
fun completeTask_dataAndSnackbarUpdated() = runTest {
    // With a repository that has an active task
    tasksRepository = NEED_NEW_CONTEXT
    tasksViewModel = NEED_NEW_CONTEXT

    val task = Task(id = "id", title = "Title", description = "Description")
    tasksRepository.addTasks( ...tasks = task)

    // Complete task
    tasksViewModel.completeTask(task,  completed = true)

    // Verify the task is completed
    assertThat(tasksRepository.savedTasks.value[task.id]?.isCompleted).isTrue()

    // The snackbar is updated
    assertThat(tasksViewModel.uiState.first().userMessage)
        .isEqualTo("Task marked complete")
}
```

Unit Testing TasksViewModel: Solution

- **Fake:** create a class `FakeTaskRepository` & `FakeTasksViewModel` with dummy functions
  - Large cost
  - Can’t apply to Android framework classes (like `LiveData`)
- **Mock & Stub:** mock `FakeTaskRepository` & `FakeTasksViewModel` using *Mockito*!

Unit Testing TasksViewModel: Fake (1/4)

```
class TasksViewModelTest {

    // Subject under test
    32 Usages
    private lateinit var tasksViewModel: TasksViewModel

    // Use a fake repository to be injected into the viewmodel
    5 Usages
    private lateinit var tasksRepository: FakeTaskRepository

    @Before
    fun setupViewModel() {
        // We initialise the tasks to 3, with one active and two completed
        val tasksRepository = FakeTaskRepository()
        val task1 = Task(id = "1", title = "Title1", description = "Desc1")
        val task2 = Task(id = "2", title = "Title2", description = "Desc2", isCompleted = true)
        val task3 = Task(id = "3", title = "Title3", description = "Desc3", isCompleted = true)
        tasksRepository.addTasks( ...tasks = task1, task2, task3)


        tasksViewModel = TasksViewModel( taskRepository = tasksRepository, SavedStateHandle())
    }
```

1. Create and initialize fake objects

2. Declare our system under test (`TasksViewModel`)

Unit Testing TasksViewModel: Fake (2/4)

```
@Test
fun completeTask_dataAndSnackbarUpdated() = runTest {
    // With a repository that has an active task
    val task = Task(id = "id", title = "Title", description = "Description")
    tasksRepository.addTasks( ...tasks = task)

    tasksViewModel.completeTask(task,  completed = true)

    assertThat(tasksRepository.savedTasks.value[task.id]?.isCompleted).isTrue()

    assertThat(tasksViewModel.uiState.first().userMessage)
        .isEqualTo("Task marked complete")
}
```

3. Write a fake implementation for every member functions

Unit Testing TasksViewModel: Fake (3/4)

4. Check `FakeTaskRepository.kt` and check its complexity!

Compare it with `DefaultTaskRepository.kt`

Unit Testing TasksViewModel: Fake (4/4)

Fake vs DefaultTaskRepository

- All member function implemented
- **Behavior mismatch**: It skips concurrency, dispatcher usage, and data conversions (toLocal, toNetwork), so it can't validate real synchronization or data flow
- **No concurrency realism**: Running purely in-memory and synchronously hides race conditions, cancellation, and backpressure issues that appear in production

→ Let's explore others!

Unit Testing TasksViewModel: Mock

1. Create and initialize mocks

```
@RunWith( value = MockitoJUnitRunner::class)
class TasksViewModel_Simple_Test {

    5 Usages
    @Mock lateinit var taskRepository: DefaultTaskRepository

    @Test(expected = NullPointerException::class)
    fun constructingAndCollecting() = runTest {
        val vm = TasksViewModel(taskRepository, SavedStateHandle())

        vm.uiState.first()

        vm.completeTask(Task( title = "1",  description = "t",  isCompleted = false,  id = "id"), completed = true).join()

    }
```

annotations added

Unit Testing TasksViewModel: Mock

2. Declare our system under test (`TasksViewModel`)

```
@RunWith( value = MockitoJUnitRunner::class)
class TasksViewModel_Simple_Test {

    5 Usages
    @Mock lateinit var taskRepository: DefaultTaskRepository

    @Test(expected = NullPointerException::class)
    fun constructingAndCollecting() = runTest {
        val vm = TasksViewModel(taskRepository, SavedStateHandle())

        vm.uiState.first()

        vm.completeTask(Task( title = "1",  description = "t",  isCompleted = false,  id = "id"), completed = true).join()

    }
```

Uses the original class instead of fake

Creating and Initializing Mocks

1. Create a mock
   - `@Mock` annotation ← we used this
   - `mock()` function
2. Initialize mocks
   - `@RunWith(MockitoJUnitRunner.class)` ← we used this
   - MockitoAnnotations.openMocks(this)

Unit Testing TasksViewModel: Mock (1/5)

- Create a class `TasksViewModel_Simple_Test`
- First test `TasksViewModel` initialization part

```
@Test
fun constructingAndCollecting()= runTest{
    // GIVEN a fresh taskRepository (already mocked: TODO)
    // WHEN a viewModel is initialized and completeTask called
    val vm = TasksViewModel(taskRepository, SavedStateHandle())
    vm.uiState.first()
    vm.completeTask(Task("1", "t", false, "id"), completed =
true).join()

    // THEN UiState is changed
    //...
}
```

Unit Testing TasksViewModel: Mock (2/5)

- Test Result: SUCCESS

Run TasksViewModel_Simple_Test.constructingAndCollectin...

Test Results 264 ms

1 test passed 1 test total, 264 ms

```
Executing tasks: [:app:testDebugUnitTest, --tests, com.example.android.architecture.blueprints.todoapp.tasks.TasksViewModel_Simple_Test.co

Reusing configuration cache.
> Task :shared-test:preBuild UP-TO-DATE
> Task :app:preBuild UP-TO-DATE
> Task :shared-test:checkKotlinGradlePluginConfigurationErrors SKIPPED
> Task :app:checkKotlinGradlePluginConfigurationErrors SKIPPED
> Task :shared-test:preDebugBuild UP-TO-DATE
```

Unit Testing TasksViewModel: Mock (3/5)

- Let's test one more:

```
class TasksViewModel @Inject constructor(
    private val taskRepository: TaskRepository,
    private val savedStateHandle: SavedStateHandle
) : ViewModel() {

    2 Usages
    private val _savedFilterType =
        savedStateHandle.getStateFlow( key = TASKS_FILTER_SAVED_STATE_KEY,  initialValue = ALL_TASKS)

    1 Usage
    private val _filterUiInfo = _savedFilterType.map { getFilterUiInfo( requestType = it) }.distinctUntilChanged()
    3 Usages
    private val _userMessage: MutableStateFlow<Int?> = MutableStateFlow( value = null)
    3 Usages
    private val _isLoading = MutableStateFlow( value = false)
    1 Usage
    private val _filteredTasksAsync =
        combine( flow = taskRepository.getTasksStream(),  flow2 = _savedFilterType) { tasks, type ->
            filterTasks(tasks,  filteringType = type)
        }
            .map { Async.Success( data = it) }
            .catch<Async<List<Task>>> { emit( value = Async.Error("Error while loading tasks")) }

    1 Usage
    val uiState: StateFlow<TasksUiState> = combine(
        flow = _filterUiInfo,  flow2 = _isLoading,  flow3 = _userMessage,  flow4 = _filteredTasksAsync
    ) { filterUiInfo, isLoading, userMessage, tasksAsync ->
        when (tasksAsync) {
```

Would this work well?

Unit Testing TasksViewModel: Mock (4/5)

- Let's test one more: add a line
  - `println("FLOW TYPE >>> ${taskRepository.getTasksStream()::class}")`

```
@RunWith( value = MockitoJUnitRunner::class)
class TasksViewModel_Simple_Test {

    2 Usages
    @Mock lateinit var taskRepository: DefaultTaskRepository

    @Test
    fun constructingAndCollectingViewModel() = runTest {
        val vm = TasksViewModel(taskRepository, SavedStateHandle())
        vm.uiState.first()
        println("FLOW TYPE >>> ${taskRepository.getTasksStream()::class}")

        vm.completeTask(Task( title = "1",  description = "t",  isCompleted = false,  id = "id"), completed = true).join()

    }
```

Unit Testing TasksViewModel: Mock (5/5)

- Test Result: FAILED..

```
1 test failed   1 test total, 250 ms
WARNING: Dynamic loading of agents will be disallowed by default in a future release

Cannot invoke "Object.getClass()" because the return value of "com.example.android.architecture.blueprints.todoapp.data
 .DefaultTaskRepository.getTasksStream()" is null
java.lang.NullPointerException Create breakpoint : Cannot invoke "Object.getClass()" because the return value of "com.example.android
 .architecture.blueprints.todoapp.data.DefaultTaskRepository.getTasksStream()" is null
    at com.example.android.architecture.blueprints.todoapp.tasks.TasksViewModel_Simple_Test$constructingAndCollectingViewModel$1
     .invokeSuspend(TasksViewModelTest.kt:41)
    at com.example.android.architecture.blueprints.todoapp.tasks.TasksViewModel_Simple_Test$constructingAndCollectingViewModel$1.invoke
     (TasksViewModelTest.kt)
    at com.example.android.architecture.blueprints.todoapp.tasks.TasksViewModel_Simple_Test$constructingAndCollectingViewModel$1.invoke
     (TasksViewModelTest.kt) <1 internal line>
    at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl.kt:33) <3 internal lines>
<61 folded frames>
```

- Why? → We need to implement some more details

Unit Testing TasksViewModel: Stub (1/4)

- `@Mock` creates a mock instance that implements the functions of `TasksRepository`
- *Unstubbed* methods return **default values (reference types ⇒ null)**
  - Other mocked functions are expected to return null as default
  - Mocked `getTasksStream()` returns `null` (!= default value “`[]`”)
- So we should ***stub*** these functions

AI 개요

+9

"stub"의 사전적 의미는 여러 가지가 있습니다. 주로 사용되는 의미는 토막, 꼬리, 동강, 조각 등을 뜻하는 명사입니다. 또한, 프로그래밍 분야에서는 테스트를 위해 실제 기능 대신 사용하는 임시 코드를 의미하기도 합니다.

1. 명사로서의 의미:

Unit Testing TasksViewModel: Stub (2/4)

- `uiState` depends on `TaskRepository.getTasksStream()` (via `combine`).
- To test `completeTask()`, we must stub `getTasksStream()` before creating the `ViewModel` so that collecting `uiState` works.

```
@Before
fun setUp() {
    Mockito.`when`( methodCall = taskRepository.getTasksStream()).thenReturn( value = MutableStateFlow( value = emptyList()))
    vm = TasksViewModel(taskRepository, SavedStateHandle())
}

@Test
fun constructingAndCollectingViewModel() = runTest {

    vm.uiState.first()
```

Add this line

Unit Testing TasksViewModel: Stub (3/4)

```
@RunWith(MockitoJUnitRunner::class)
class TasksViewModel_With_Stub_Test {

    private lateinit var vm: TasksViewModel

    @Mock lateinit var taskRepository: DefaultTaskRepository

    @Before
    fun setUp() {
        Mockito.`when`(taskRepository.getTasksStream()).thenReturn(MutableStateFlow(emptyList()))
        vm = TasksViewModel(taskRepository, SavedStateHandle())
        println("FLOW TYPE >>> ${taskRepository.getTasksStream()::class}")
    }

    @Test
    fun constructingAndCollectingViewModel() = runTest {

        vm.uiState.first()
        vm.completeTask(Task("1", "t", false, "id"), completed = true).join()

    }
}
```

Unit Testing TasksViewModel: Stub (4/4)

- Then it works

```
ns  ✓ 1 test passed  1 test total, 274 ms
> Task :app:testDebugUnitTest
WARNING: A Java agent has been loaded dynamically (/Users/sieun/.gradle/caches/8.11.1/transforms/72ea0f62f313aa700c4c07b69bc9cd48/transfor
WARNING: If a serviceability tool is in use, please run with -XX:+EnableDynamicAgentLoading to hide this warning
WARNING: If a serviceability tool is not in use, please run with -Djdk.instrument.traceUsage for more information
WARNING: Dynamic loading of agents will be disallowed by default in a future release
FLOW TYPE >>> class kotlinx.coroutines.flow.StateFlowImpl (Kotlin reflection is not available)
class kotlinx.coroutines.flow.StateFlowImpl (Kotlin reflection is not available)
TasksViewModel_With_Stub_Test > constructingAndCollectingViewModel PASSED
BUILD SUCCESSFUL in 1s
49 actionable tasks: 1 executed, 48 up-to-date
Configuration cache entry reused.
```

Stubbing: Functions with Return Values

- `when().thenReturn()`
  - To specify a return value when called with specific params
- `when().thenThrow()`
  - To throw exceptions when function is called
- `when().thenAnswer()`
  - Allows stubbing with the generic Answer interface
  - Don’t recommend using use thenReturn() or thenThrow()

Stubbing: void Functions (1/2)

- `doThrow()`: to stub a void method with an exception
- `doAnswer()`: to stub a void method with generic Answer
- `doNothing()`: for void methods to do nothing (which is default)
- `doReturn()`: when you cannot use when(Object)
- `doCallRealMethod()`: to call the real implementation of a method

Stubbing: void Functions (2/2)

```
// Stubbing void function
doThrow(new RuntimeException()).when(mockedObject).voidFunction()
// NOT doThrow(...).when(mockedObject.voidFunction)
mockedObject.voidFunction() // throws RuntimeException:
```

- `doSomething().when(mockedObject).function()` format:
  - Also used for:
    - stubbing methods on spy objects
    - stubbing the same method more than once, to change behavior in the middle of a test

StateFlow

- Always holds the latest value → a Hot Flow
- UI collects new values with collect { ... }
- Requires an initial value when created
- Value updates via .value (similar to LiveData.setValue())
- **Compared to LiveData**
  - LiveData: lifecycle-aware (Android only) / Tightly coupled with UI layer
  - **StateFlow**: coroutine-based, lifecycle handling must be explicit
  - Compose-friendly - our repo also has compose-based view

https://readystory.tistory.com/207

StateFlow

- A simple example

```
private val _isLoading = MutableStateFlow(false)
val isLoading: StateFlow<Boolean> = _isLoading

// Update value
_isLoading.value = true

// Collect in UI
lifecycleScope.launch {
    viewModel.isLoading.collect { loading ->
        showLoading(loading)
    }
}
```

Testing StateFlow

- Add dependency: turbine (`build.gradle.kts (:app)`)

```
dependencies {
    testImplementation(libs.androidx.compose.ui.test.junit)
    testImplementation("org.mockito:mockito-core:5.11.0")
    testImplementation("app.cash.turbine:turbine:1.1.0")
```

Unit Testing TasksViewModel: StateFlow (1/6)

- Let's test **`completeTask(), clearCompleteTasks()`** now
- See `TasksViewModel_StateFlow_Test()` in `TasksViewModelTest.kt` (partially implemented)
  → uncomment it

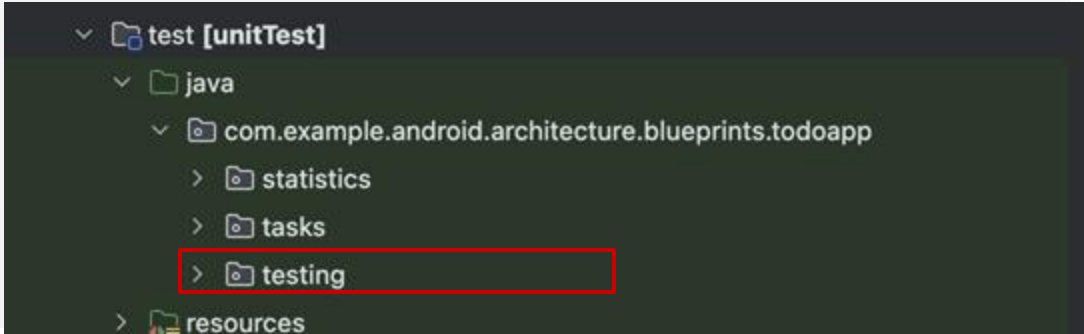
link

Unit Testing TasksViewModel: StateFlow (2/6)

1. To test StateFlow, we should define a dispatcher rule
   - Add `testing/MainDispatcherRule.kt`

```
class MainDispatcherRule(
    private val dispatcher: TestDispatcher = StandardTestDispatcher()
) : TestRule {
    override fun apply(base: Statement, description: Description): Statement =
        object : Statement() {
            override fun evaluate() {
                Dispatchers.setMain(dispatcher)
                try {
                    base.evaluate()
                } finally {
                    Dispatchers.resetMain()
                }
            }
        }
}
```

Take your time fixing import errors



Unit Testing TasksViewModel: StateFlow (3/6)

- Then add rule to our test
  - `UnconfinedTestDispatcher`: for performance

```
class TasksViewModel_StateFlow_Test {

    @get:Rule val main = MainDispatcherRule(UnconfinedTestDispatcher()) // or StandardTestDispatcher()
```

Unit Testing TasksViewModel: StateFlow (4/6)

2. Stub functions (e.g., `clearCompletedTasks()`):

```
Mockito.doAnswer { inv ->
      val id = inv.arguments[0] as String
            tasksFlow.value = tasksFlow.value.filterNot { it.isCompleted }
      Unit
}.`when`(repo).clearCompletedTasks()
```

- This is needed to check if function `completeTask()`, `clearCompleteTasks()` actually react!

Unit Testing TasksViewModel: StateFlow (5/6)

3. Write a test

```
@Test
    fun clearCompletedTasks_removesCompleted_andSnackbar() = runTest {
        vm.uiState.test {
            val initial = awaitItem() // Waits and checks the value StateFlow emits

            vm.clearCompletedTasks()
            runCurrent()

            val s1 = awaitItem()
            assertEquals(R.string.completed_tasks_cleared, s1.userMessage)

            val s2 = awaitItem()
            assertTrue(s2.items.none { it.isCompleted })

            cancelAndIgnoreRemainingEvents()
        }
    }
```

Unit Testing TasksViewModel: StateFlow (6/6)

4. Check if it works

* Comment out these lines for now!

(to prevent unnecessary stub warning)

```
Mockito.`when`(methodCall = repo.getTasksStream()).thenReturn(value = tasksFlow)

//        // suspend stubs must return Unit (not null!)
//        Mockito.doAnswer { inv ->
//            val id = inv.arguments[0] as String
//            tasksFlow.value = tasksFlow.value.map { if (it.id == id) it.copy(isCompleted = false) else it }
//            Unit
//        }.`when`(repo).activateTask(Mockito.anyString())

Mockito.doAnswer {
```

3 mins

Exercise 3: Stub&Test More Functions

Stub the `completeTask()` function

: in the same way as the `activateTask()` stub in the skeleton code

Keep stub of `activateTask()` commented out when running this test!

7 mins

Exercise 4: Testing `completeTask()`

- Verify that marking a completed task as active updates both snackbar and list (Fill TODO in `completeTask_false_updatesSnackbar_andList()`)
- Hint
  - 1. Launch a no-op collector on vm.uiState.
  - 2. Call vm.completeTask(task, completed = false).
  - 3. Run the scheduler (runCurrent() / advanceUntilIdle()).
  - 4. Assert:
    - Snackbar message = R.string.task_marked_active.
    - Task with id "2" is active in uiState.value.items.
  - 5. Cancel the collector.

7 mins

Exercise 4: Testing `completeTask()`

IMPORTANT

- Comment out the `completeTask()` stub here
- Uncomment the `activateTask()` stub

to prevent unnecessary stub errors

Unit Testing for `TasksViewModel`: Recap

- Created **mock** for external dependency `taskRepository`
- Made `getTasksStream()` return specific values (**stub**)
- Checked that uiState's value has changed to these values (**observe StateFlow change**)

Submission

- Final code: `TasksViewModel_Final_Test` class
- Copy `TasksViewModelTest.kt` into the `submission` directory, under both `exercise03` and `exercise04` (adjusting it to each exercise's requirements)
- Make sure all tests pass successfully

Mockito: Additional Information

- Mockito `verify()`
- Argument matching
- Spy
- InjectMock

Mockito verify()

- We can use Mockito `verify()` to:
  - Check the exact number of invocations, redundant invocations
  - Check in-order calls

Mockito verify(): Number of Calls

```
//using mock
mockedList.add("once")

mockedList.add("three times")
mockedList.add("three times")
mockedList.add("three times")
```

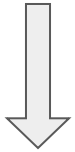
```
//exact number of invocations verification
verify(mockedList).add("once")
verify(mockedList, times(3)).add("three times")

//verification using never(). never() is an alias to times(0)
verify(mockedList, never()).add("never happened")

//verification using atLeast()/atMost()
verify(mockedList, atMostOnce()).add("once")
verify(mockedList, atLeast(2)).add("three times")
```

Mockito verify(): In-Order Calls (1/2)

```
val firstMock = mock<List<String>>()
val secondMock = mock<List<String>>()

//using mocks
firstMock.add("was called first")
firstMock.add("was called second")
secondMock.add("was called third")
secondMock.add("was called fourth")

//create inOrder object by passing any mocks that
need to be verified in order
val inOrder = inOrder(firstMock, secondMock)
```

If order is right:

```
//verifications to assure input order
inOrder.verify(firstMock).add("was called first")
inOrder.verify(firstMock).add("was called second")
inOrder.verify(secondMock).add("was called third")
inOrder.verify(secondMock).add("was called fourth")
```



Test passes!

```
✔ Tests passed: 1 of 1 test – 1 sec 126 ms
Executing tasks: [:app:testDebugUnitTest, --tests, com.example.la
```

Mockito verify(): In-Order Calls (2/2)

```
val firstMock = mock(List::class.java) as MutableList<String>
val secondMock = mock(List::class.java) as MutableList<String>


//using mocks
firstMock.add("was called first")
firstMock.add("was called second")
secondMock.add("was called third")
secondMock.add("was called fourth")

//create inOrder object by passing any mocks that need to
verified in order
val inOrder = inOrder(firstMock, secondMock)
```

If order is not right:

```
//using mocks
secondMock.add("was called fourth")
secondMock.add("was called third")
firstMock.add("was called first")
firstMock.add("was called second")
```

Error!

```
Tests failed: 1 of 1 test – 965 ms
Verification in order failure
Wanted but not invoked:
list.add("was called third");
-> at com.example.lab4application.ExampleUnitTest.succeedingTest(ExampleUnitTest.java:52)
Wanted anywhere AFTER following interaction:
list.add("was called second");
-> at com.example.lab4application.ExampleUnitTest.succeedingTest(ExampleUnitTest.java:43)
```

Argument Matchers

- Allows flexible stubbing and verification
- Common ArgumentMatchers:
  - Exact Value Matchers: `eq(value), any(), isNull()...`
  - Generic Matchers: `any()`
  - Numeric Matchers: `any<Int>(), any<Long>(), any<Double>()...`
  - Collection Matchers: `any<List<*>>(), any<Set<*>>(), any<Map<*>>()...`
-

```
whenever(mockedList.get(any())).thenReturn("element")

println(mockedList[999])

verify(mockedList).get(any())
```

Stubbing with ArgumentMatcher

Verifying with ArgumentMatcher

Source: https://javadoc.io/static/org.mockito/mockito-core/5.5.0/org/mockito/ArgumentMatchers.html

Spying with Mockito

- Spying allows you to use real objects and selectively stub certain methods
- Use the **`@Spy`** annotation or **`spy()`** method
- Recommended to use **`doReturn/Answer/Throw()`** for stubs (sometimes `when().thenReturn()` won't work)

Spying with Mockito

- Mockito does NOT delegate calls to the passed real instance, instead it creates a *copy* of it.
  - If you keep the real instance and interact with it, the spied object won't be affected
  - If an un-stubbed method is called on the *spy* but not on the real instance, there is NO effect on the real instance

Spying with Mockito: Example

- Use real method `ArrayList.add()`
- Use stubbed method `ArrayList.size()`

```
@Test
fun failingTest() {
    val list = LinkedList<String>()
    val spy = spy(list)

    // Real method is called
    // IndexOutOfBoundsException is thrown (bc list is empty)
    spy.get(0)
    assertEquals("foo", spy.get(0))
}
```

```
@Test
fun failingTest() {
    val list = LinkedList<String>()
    val spy = spy(list)

    // You have to use doReturn() for stubbing
    doReturn("foo").`when`(spy).get(0)
    assertEquals("foo", spy.get(0))
}
```

Test fails…

```
Index: 0, Size: 0
java.lang.IndexOutOfBoundsException Create breakpoint : Index: 0, Size: 0
    at java.base/java.util.LinkedList.checkElementIndex(LinkedList.java:559)
    at java.base/java.util.LinkedList.get(LinkedList.java:480)
    at com.example.lab4application.ExampleUnitTest.failingTest(ExampleUnitTest.java:69) <37 internal lines>
    at jdk.proxy1/jdk.proxy1.$Proxy2.processTestClass(Unknown Source) <7 internal lines>
    at worker.org.gradle.process.internal.worker.GradleWorkerMain.run(GradleWorkerMain.java:69)
    at worker.org.gradle.process.internal.worker.GradleWorkerMain.main(GradleWorkerMain.java:74)
```

Test passes!

Mock vs Spy

- `@Mock` creates a fake class with empty functions. We need to stub functions else will return null
- `@Spy` creates an actual class. By default, functions maintain their functionality, and only stubbed ones are overridden
- Spies should be used carefully and occasionally
  - Real methods are invoked → unexpected results

@InjectMocks

- Automatically injects mock or spy dependencies into tested objects
- Simplifies process of injecting mocks into tested objects

```
class MyDictionary {
    private val wordMap: MutableMap<String, String> = HashMap()

    fun add(word: String, meaning: String) {
        wordMap[word] = meaning
    }
    fun getMeaning(word: String): String? {
        return wordMap[word]
    }
}
```

```
class MyDictionaryTest {
    @Mock
    lateinit var wordMap: MutableMap<String, String>

    @InjectMocks
    lateinit var dic : MyDictionary

    @Test
    fun whenUseInjectMocksAnnotation_thenCorrect() {
        Mockito.`when`(wordMap["aWord"]).thenReturn("aMeaning")
        assertEquals("aMeaning", dic.getMeaning("aWord"))
    }
}
```

Automatically injected

Can stub the injected mock

Mockito Other Features

- Visit official documentation:
  - Resetting mocks
  - Using Mockito for Behavior Driven Development (BDD)

For more…

- Highly recommend reading:
  - Testing Basics codelab: full app tested, good practices in Android
  - Official Android Studio Test doc:
    - Configuration for advanced testing
    - Other types of testing (monkey test…)
  - Official Android Test doc:
    - Details & Extensions on what we've done today
    - Good practices

Hilt

- A tool to help dependency injection in Android
- Object creation and scoping, and lifecycle is automatically handled by Hilt!
- Easily integrated with Android Jetpack
- Standardized dependency injection logic
  → easy to collaborate

Hilt

```
// 의존성: Engine
class Engine @Inject constructor() {
    fun start() { println("Engine started") }
}

// Engine을 필요로 하는 Car
class Car @Inject constructor(private val engine: Engine) {
    fun drive() {
        engine.start()
        println("Car is moving!")
    }
}

// MainActivity에서 사용
@AndroidEntryPoint
class MainActivity : AppCompatActivity() {
    @Inject lateinit var car: Car

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        car.drive()
    }
}
```

When creating a Car, Hilt automatically creates and injects the required Engine.

In MainActivity, you can simply use the car variable directly.

Using Hilt

- Add dependencies

```
plugins {
  ...
  id("com.google.dagger.hilt.android") version "2.56.2" apply false
}
```

```
plugins {
  id("com.google.devtools.ksp")
  id("com.google.dagger.hilt.android")
}

android {
  ...
}

dependencies {
  implementation("com.google.dagger:hilt-android:2.56.2")
  ksp("com.google.dagger:hilt-android-compiler:2.56.2")
}
```

Using Hilt

```
// enabling Hilt
@HiltAndroidApp
class MyApplication : Application() { ... }
```

```
// provide instances with constructor
class AnalyticsAdapter @Inject constructor() { ... }
```

```
// enable Hilt in the activity
@AndroidEntryPoint
class MainActivity : AppCompatActivity() {
  @Inject lateinit var analytics: AnalyticsAdapter
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // analytics instance has been populated by Hilt
    // and it's ready to be used
  }
}
```

https://medium.com/androiddevelopers/dependency-injection-on-android-with-hilt-67b6031e62d

Hilt

- See this doc for details:
  - https://developer.android.com/training/dependency-injection/hilt-android
  - https://medium.com/androiddevelopers/dependency-injection-on-android-with-hilt-67b6031e62d
- Hilt is mostly implemented in the provided source code!

```
19 Usages
@HiltViewModel
class AddEditTaskViewModel @Inject constructor(
    private val taskRepository: TaskRepository,
    savedStateHandle: SavedStateHandle
) : ViewModel() {

    5 Usages
```

Summary

- Next week: Integration tests!
  - UI testing with Espresso
  - Espresso basics
  - Test coverage reports
  - Good practices when writing tests

Exercise Submission

- Due: 2025/10/24 Fri 23:59
- Submit exercise01 - exercise04 in a **single zip file:**
  - File structure
    - submission
      - exercise01
        - // files
      - exercise02
        - // files
      - exercise03
        - // files
      - exercise04
        - // files

Thank You.
Any Questions?