# Guided Learning (QLC-2) — Mood Tag Selector Extension (With TDD & Mockito Edition)

### Objective

Extend the **Mood Tag Selector** app by introducing **batch analytics** using **Test-Driven Development (TDD)** and an **interface-driven architecture**.

### Initial Setup (QLC-1.1)

* We created a new **Android project** with an appropriate package structure.
* We implemented the MoodTagSelectorViewModel class.
* We added basic logic to store selected moods (e.g., "Happy", "Sad").
* We wrote a unit test to verify that selecting a mood (like "Happy") correctly adds it to the list.

### Extension Requirement (QLC-2)

The project now needs to support **batch mood tagging** for datasets (such as daily logs).

To meet this requirement:

* We introduced a MoodAnalyticsManager class that collaborates with MoodTagSelectorViewModel.
* The MoodAnalyticsManager is responsible for applying mood logic to **collections** (e.g., logs, timestamps, categories).

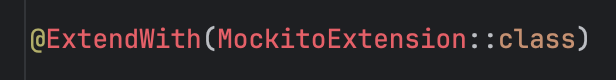
This design keeps responsibilities clearly separated and ensures each component remains easy to test.

### Test Setup Reminder:

Before writing your tests, ensure you have the proper **Mockito setup** and **test environment configuration**.  
This guarantees that:

* Dependencies are correctly mocked
* Your test classes behave consistently and predictably
* You avoid issues with uninitialized mocks or unexpected behavior

In **JUnit 5 (Jupiter)**, this setup is typically handled by annotating your test class with:



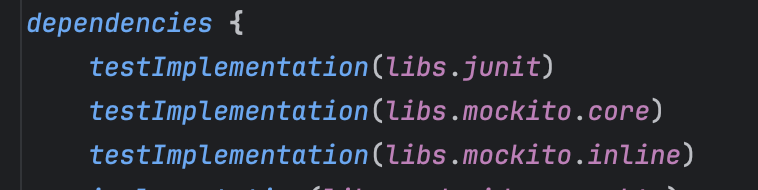
This tells Mockito to:

* Automatically create and inject mocks for any fields annotated with @Mock
* Manage the lifecycle of mocks (no manual initialization required)

When using **JUnit 4** with **Mockito**, you must use the @RunWith(MockitoJUnitRunner::class) annotation on your test class.

Similarly to the above, this instructs Mockito to:

* Automatically initialize all fields annotated with @Mock
* Manage the lifecycle of your mocks for each test case
* Remove the need for manual mock initialization

Additionally, we’ll need the following dependencies:  


# 

### Reminder: Red → Green → Refactor (TDD in QLC-2)

In this exercise, we’re applying **Test-Driven Development** to a system with **multiple components**.

* **Red:** Start by writing a test that fails, it defines what we want the class to do.
* **Green:** Implement the bare minimum to make that test pass. No more.
* **Refactor:** Improve the code (e.g. extract interfaces, rename things, make it more modular) without changing what it does.

### Step 1 — Define the Feature Behavior and Requirements

In **QLC-2**, we are extending the Mood Tag Selector feature to support **batch mood analytics**.  
The focus is no longer just on single mood selection but on processing collections of moods. For example, analyzing mood patterns across daily logs or user sessions.

We want a backend feature (not a direct UI feature) that processes a list of mood tags using batch logic.

This feature must:

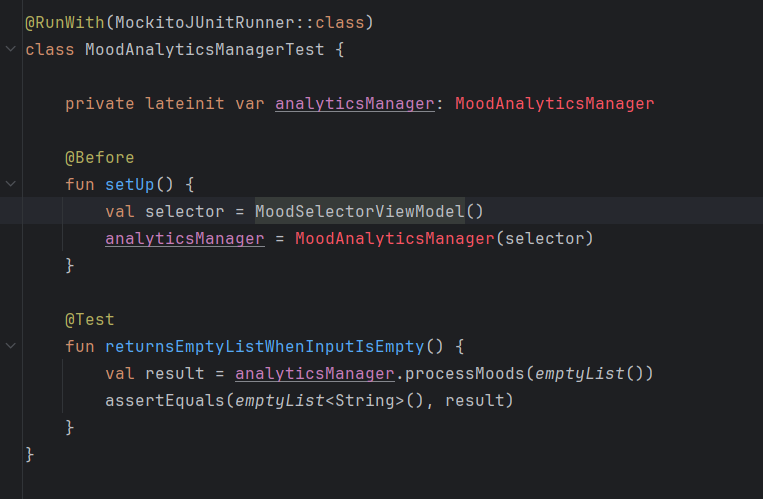
* Accept a collection of mood tags (e.g., "Happy", "Tired", "Motivated").
* Validate each mood tag using predefined rules (delegated to the MoodTagSelectorViewModel or TagSelectorinterface).
* Filter out invalid or empty mood tags.
* Return the processed list of valid mood tags.
* Keep this logic isolated in MoodAnalyticsManager.

### Step 2 — Define the Behavior with a Failing Unit Test (RED)

We now define the first test that expresses the batch mood processing behavior. We are starting with the simplest possible case, an **empty input list**.

The expected behavior:

* When MoodAnalyticsManager processes an empty list of moods, it returns an empty list.

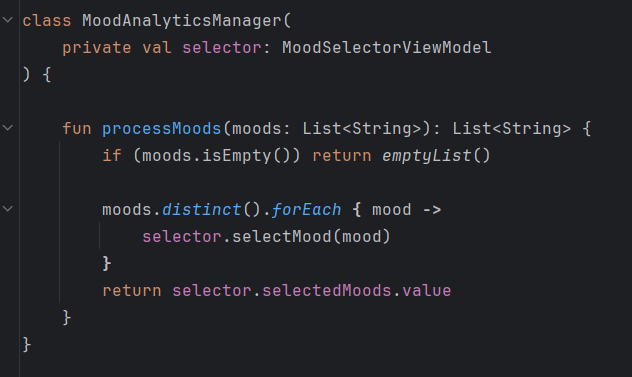


We want this test to fail initially because the implementation does not yet exist.

### Step 3 — Implement the Simplest Working Logic (GREEN)

Now that your unit test (returns empty list when input is empty) is red, write the *smallest* amount of code to make it pass.

1. **Create** a MoodAnalyticsManager class in **your main source set** that takes a MoodTagSelectorViewModel in its constructor.
2. **Inside** processMoods(moods: List<String>): List<String>:  
   * If moods is empty, immediately return emptyList().
   * Otherwise, loop through moods.distinct().
   * For each mood, call selector.selectMood(mood) on your MoodTagSelectorViewModel.
   * After the loop, return the ViewModel’s selectedMoods list.
3. (For pure logic) use a simple mutableListOf<String>() internally, or rely on the ViewModel’s own list.



## QLC-2.1: Testing Structure Analysis (REFACTOR)

### Objective

Extract your local‑state selection logic out of the manager and into a TagSelector interface implemented by your ViewModel. Update your tests to mock the interface so that **all existing tests remain green**.

### Why Refactor?

* **Fragility:** Right now MoodAnalyticsManager depends directly on MoodTagSelectorViewModel—when the ViewModel changes, your manager tests break.
* **Complexity:** You’re forced to pull in the entire ViewModel (and its StateFlow) just to test pure batch‑processing logic.
* **Blurred Focus:** Tests for MoodAnalyticsManager need only verify its filtering/aggregation behavior, not any UI or state‑management code.

Extracting a simple TagSelector interface restores Single Responsibility and makes your tests laser‑focused.

### Exercise Steps

1. **Define the TagSelector Interface** Create a new file TagSelector.kt in your main source set:
2. **Update the ViewModel to Implement TagSelector**
   1. Change MoodTagSelectorViewModel to implement this interface
3. **Change MoodAnalyticsManager to Depend on TagSelector**
   1. Modify its constructor and implementation to use the interface
4. **Rewrite Your Manager Tests to Mock TagSelector**
   1. Replace the setup that injected a real ViewModel with a Mockito mock of the interface:

Check: <https://github.com/Edrzapi/Android_tdd_tag_selector/tree/QLC-1.1-ANS> out for the solution

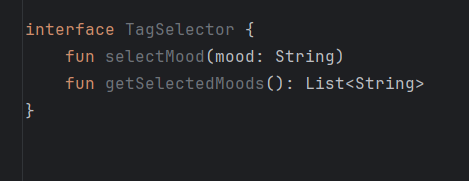
**QLC-2.1: Solution - Testing Structure Analysis**

### Introduce Interface for Decoupling

Although our current **Green** implementation of MoodAnalyticsManager passes the empty‑list test, it is tightly coupled to MoodTagSelectorViewModel. This coupling has two drawbacks:

1. **Responsibility Creep** MoodAnalyticsManager now depends on the ViewModel’s internal state management, mixing batch‑processing logic with UI‑state concerns.
2. **Test Fragility** Any change to MoodTagSelectorViewModel—even unrelated to analytics—will break our manager tests.

To restore a clear separation of concerns and improve testability, we extract a simple interface:



#### Refactor Steps

1. **Rewrite manager tests** to mock TagSelector rather than instantiate the ViewModel directly.
2. **Define TagSelector** in your main source.
3. **Update MoodTagSelectorViewModel** to implement TagSelector, delegating its existing state logic.
4. **Change MoodAnalyticsManager’s constructor** to accept a TagSelector instead of a concrete ViewModel.

After this refactor:

* MoodAnalyticsManager focuses solely on batch‑processing logic.
* Tests for the manager mock TagSelector, isolating analytics behavior from state management.
* The ViewModel continues to drive UI state without leaking into analytics tests.

This decoupling aligns with the Single Responsibility and Dependency Inversion principles, making our codebase cleaner, more modular, and easier to maintain.

This interface allows us to **decouple** analytics from the ViewModel, and makes it easier to **mock**

in unit tests.

## Step 1: Updating the Test

## Step 3: Updating the ViewModel

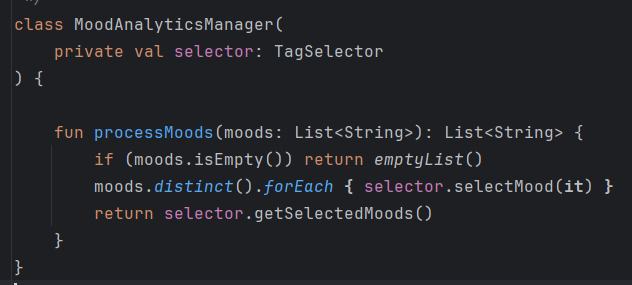
Now that we’ve extracted the TagSelector interface, we need our ViewModel to fulfill that contract. This lets MoodAnalyticsManager depend on the interface rather than a concrete implementation.



#### Why These Choices?

* **MutableStateFlow / StateFlow** We use a StateFlow to model mutable UI state in a reactive, lifecycle‑safe way. The ViewModel updates \_selectedMoods, and any Composables observing selectedMoods will recompose automatically.
* **Immutability on Updates** By assigning \_selectedMoods.value = \_selectedMoods.value + mood, we create a new list each time. This prevents accidental in‑place mutations, ensuring StateFlow emits correctly.
* **Interface Implementation** Implementing TagSelector here allows MoodAnalyticsManager to operate against an abstraction, not a concrete class. This decoupling makes both components easier to test and evolve independently.
* **Validation Logic** Checking mood.isNotBlank() and !contains(mood) enforces the feature requirements (no empty tags, no duplicates) in a single place, so both UI and analytics share the same rules.

## Step 4: MoodAnalyticsManager



## Step 5: Verifying with Mockito

### At this point, MoodAnalyticsManager depends on the TagSelector interface. We want to ensure that, when processing a list of moods, the manager calls the correct methods on its collaborator without relying on any concrete state or UI components.

### Create a mock of TagSelector

### Invoke processMoods(...) on MoodAnalyticsManager

### Verify that selectMood(...) was called once for each unique mood

1. Implement more tests

