## Guided Learning (QLC‑1): Building the Mood Tag Selector with TDD

**Objective:**In this exercise, you will apply **Test-Driven Development (TDD)** to create a simple **Mood Tag Selector** feature using **Jetpack Compose**. You will start by writing unit tests to define the required behavior before implementing the logic.

You will write tests that ensure:

* Users can select mood tags from a predefined list.
* Selecting the same mood more than once is prevented.
* Selected moods are displayed on the screen.

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

We want a simple UI feature that allows users to select mood tags, such as *Happy*, *Tired*, or *Motivated*. When a user selects a mood:

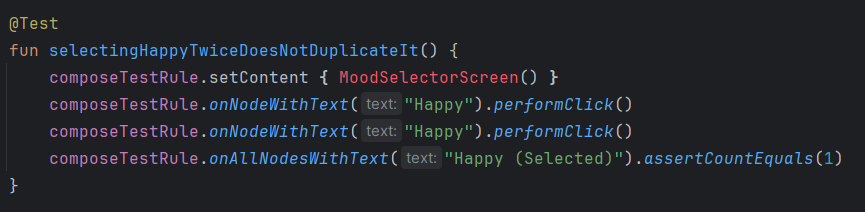
* The mood is added to a visible list of selected moods.
* The same mood **cannot** be selected again.
* The selection state is managed inside a remember state

The feature must meet these core requirements:

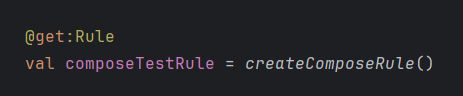
* Display a predefined list of moods.
* Selecting a mood adds it to the selected list.
* Already selected moods become disabled (cannot be selected again).
* The ViewModel controls the selection logic and exposes state to the UI.

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

Before writing any implementation, list out the exact behaviors your **MoodSelectorScreen** Composable must satisfy. Then translate each into a Compose UI test that will **fail** initially:

* **Selecting “Happy” adds it to the selected moods list.  
  **
* **Selecting “Happy” again does not add a duplicate.  
  **
* **The selected moods list updates correctly after each selection.  
  **

**Test Setup Reminder:**

Every Compose UI test class must begin by providing a ComposeTestRule. This rule handles setting up and tearing down the Compose testing environment for each test case:

Without this rule, calls to setContent { … }, onNodeWithText(), and other Compose test APIs won’t work.

**TDD Reminder:**

1. **Red:** These tests will fail until MoodSelectorScreen implements that behavior.
2. **Green:** Add just enough logic to make them pass.
3. **Refactor:** Once passing, clean up your Composable code while keeping all tests green.

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

## Now that your UI tests are failing (RED), write just enough Compose code to make them pass. In this step, you will:

## Define a local list of moods.

## Use remember { mutableStateListOf<String>() } to track selections.

## Render each mood as a button, which when clicked:

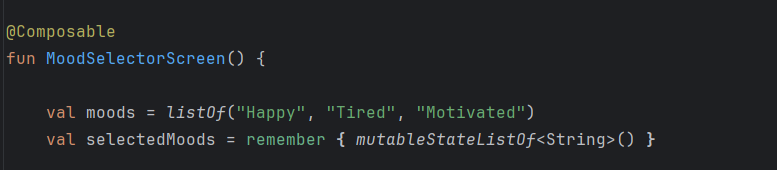
## Adds the mood to the state list if it isn’t already there.

## Disables itself once selected.

## Display all selected moods with “(Selected)” appended.

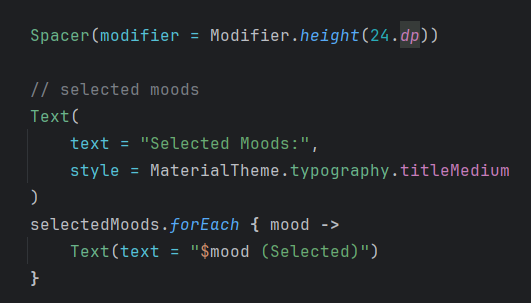
This will be the entry point for our feature. Inside it we’ll

1. Define our hard‑coded list of moods
2. Create a local, in‑Composable selection state using remember { mutableStateListOf<String>() }



Use a Column to stack our buttons vertically. For each mood we'll render a Button whose onClick adds that mood to selectedMoods **only if** it isn’t already present—and disables itself once tapped:



Below the buttons, show a header label and then loop over selectedMoods, appending “(Selected)” to each one so users can see exactly what they’ve picked:

### Step 4 — Refactor into a ViewModel (GREEN)

### Just because our local‐state version works doesn’t mean it’s maintainable or testable at scale. Let’s extract the selection logic into a ViewModel using StateFlow, then have our Composable observe that state.

**Why?**

* Logic lives in the ViewModel, not in the UI.
* We can now write **unit tests** against MoodTagSelectorViewModel.
* Composable remains a thin “view” layer.

## QLC-1.1: Refactor to a ViewModel with StateFlow (REFACTOR)

**Objective:** Extract your local‐state selection logic into a ViewModel backed by StateFlow, then update your Composable to observe and drive state from that ViewModel—while keeping all tests green.

### Steps:

1. **Create the MoodTagSelectorViewModel**
   * Define a private MutableStateFlow<List<String>> for selectedMoods and expose it as StateFlow<List<String>>.
   * Provide a moods: List<String> for your predefined moods.
   * Implement a fun selectMood(mood: String) that appends to \_selectedMoods only if it isn’t already present.
2. **Update your Composable to use the ViewModel**
   * Change MoodSelectorScreen() to take a viewModel: MoodTagSelectorViewModel = viewModel() parameter.
   * Inside, call val selectedMoods by viewModel.selectedMoods.collectAsState() and val moods = viewModel.moods.
   * For each mood button, replace your local‐state mutation with viewModel.selectMood(mood) and disable the button when mood in selectedMoods.
3. **Run and Verify Tests (GREEN)**
   * Execute your existing **Compose UI tests**—they should all pass without any changes.
   * (Optionally) Add **unit tests** for your new ViewModel API
   * Rerun everything and confirm **all** UI and unit tests still pass.
4. **Refactor & Clean Up**
   * Tidy up any code duplication, improve naming, and remove leftover remember { … } logic.
   * Ensure your ViewModel is the single source of truth for mood selection.

**QLC-1.1: Solution - ViewModel Integration with Compose**

In the solution, we added:

*implementation*(*libs*.*androidx*.*lifecycle*.*viewmodel*.*compose*)

We use this to simplify how the **ViewModel** connects with our **Compose UI**, without manual wiring or boilerplate.

**Why did we add this?**

* **Access to viewModel() in Composables**Lets us retrieve the correct ViewModel instance directly in a Composable without factories or providers.
* **Automatic Lifecycle Scoping** The ViewModel is tied to the lifecycle of the Activity or Navigation graph — meaning it **survives configuration changes** like rotations and is automatically cleaned up when no longer needed.
* **Built-in SavedStateHandle Support** If you use SavedStateHandle in your ViewModel, this dependency wires it up automatically — no custom factory required.
* **Cleaner, Testable Code** You eliminate manual ViewModelProvider calls, making your code easier to read, test, and maintain.  
   Your UI focuses on displaying state, while logic stays inside the ViewModel.

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

## Step 5: Explanation

Test-Driven Development (TDD) is a core practice in software development. We start with writing the tests first and then implement just enough code to make the tests pass.  
  
- Red: Write a failing test for a new selection  
- Green Implement the logic to pass  
- Refactor: Clean up the ViewModel and UI code

### Step 5.1 — Implementing Additional Behaviors Incrementally

TDD shines when you take **small, focused steps**. Instead of building everything at once:

* Add support for **one new mood** selection at a time.
* Write the failing test for **that** behavior first.
* Implement just enough code to make **that one** test pass.
* Refactor before moving on.

This ensures you’re always working with a **green test suite** and that each feature is backed by its own test.

## Step 6: Additional Edge Cases

As you grow your Mood Selector, think about more scenarios that could trip you up:

* **Selecting the same mood twice** (already covered)
* **Disabling** the button for selected moods
* **Order**: Are moods displayed in the order they were selected?
* **Empty or huge lists**: How does the UI behave?  
  **Persistence**: What happens on screen rotation or process death?
* **Error states**: Could selecting a mood ever throw (e.g., invalid value)?

For each new case, write a failing test, implement, then refactor.

### Conclusion

You have now walked through a full TDD cycle:

* **Defined** the behavior with failing tests (Red).
* **Implemented** the simplest code to satisfy them (Green).
* **Refactored** the logic into a ViewModel with StateFlow.

This approach gives you confidence that your **Mood Tag Selector** is correct, maintainable, and easily extensible.