## QLC7 — Mutation Testing: Mood Logic

### Objective

Experience how mutation testing (with PIT and Gradle) can improve your unit tests and help you enforce the contract of your code—using real logic from your actual codebase.

### Scenario (Optional)

Your app has important logic that your team relies on for correct behaviour, whether that’s business rules in a domain class, selection logic in a ViewModel, or an analytics function. You want your tests to *really* catch unintended changes. You’ll use TDD and PIT mutation testing to check if those tests are strong enough, and you’ll choose which class or function to focus on as a group.

### Background(What is PIT?):

* **PIT** (short for *PIT Mutation Testing*) is a tool that automatically creates *mutants*—small, artificial bugs—in your code.
* It then runs your tests to see if those mutants get “caught” (do your tests fail when the code is wrong?).
* Each mutant is a *tiny edit* to your code (like changing == to !=, removing an if-statement, or always returning true).

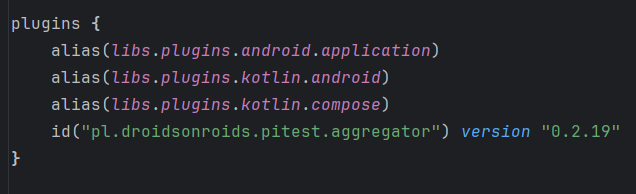
### How PIT Works: Step by Step

1. **You write your normal code and tests.**
2. **PIT takes your code** and, *one at a time*, makes a small change, a “mutation.”
   * Example:  
      If your code says if (a == b), PIT might change it to if (a != b).
3. **PIT runs your tests** after each mutation.
   * If your tests **fail**: The mutant is “killed.” Good!
   * If your tests **still pass**: The mutant “survives.” Your tests did not notice the bug.
4. **PIT reports how many mutants were killed vs. survived.**
   * More “killed” = Stronger tests.
   * “Surviving” mutants = Tests that missed possible bugs.

### Prerequisites (Test Setup):

Given occasional setup challenges, two PIT methods are provided:

* **Plugin Version (Recommended)** In your module-level build.gradle.kts add:



* Sync Gradle, then run ./gradlew pitest (FYI, the above line is the **aggregator** for multi-module projects.)
* For most single-module Android projects, you need:
  + id("info.solidsoft.pitest") version "1.15.0" - in the same position, I have included several ways due to common import problems.
* So, if that method didn’t work, try the code below . Make sure to use your IDE to highlight and add the import for the PiTestPluginExtension. Finally, run your testwith: ./gradlew pitest



### Step 1 — Identify the Core Logic (Red Phase Begins)

* Browse your codebase and **select a function or method with meaningful logic** (for example, in TagSelector, MoodAnalyticsManager, or MoodSelectorViewModel).
* Make sure it has or can have plain unit tests in src/test/java/....
* **Document or copy the logic you chose** in your lab notes or report.

### Step 2 — Write an Initial Failing Test (Red)

* Write a new unit test that would fail until the target logic is correctly implemented, or
* Pick an existing test and comment on its intention.

### Step 3 — Expand Your Unit Tests (Green)

* Write or review tests to cover all the main scenarios for your chosen function (success path, edge cases, nulls, etc).
* Aim to increase code coverage, but also logical coverage think about how someone might accidentally break the contract.

### Step 4 — Run PIT Mutation Testing

* Run mutation testing for your module

### Step 5 — Review & Analyse Surviving Mutants

* Identify any mutants that survived your tests. Examples might include:  
  + Changing conditions (e.g., == to !=)
  + Removing branches (e.g., always skipping an if/else)
  + Returning default values regardless of logic
* Reflect: Did your tests catch all changes? Any surprises?

### Step 6 — Strengthen Your Tests

* Improve or add tests to kill any surviving mutants:  
  + Add edge cases
  + Cover alternative code paths
  + Try unexpected or invalid inputs
* Re-run PIT and repeat until your mutation score is satisfactory.

### Step 7 — Reflect & Discuss

* Which mutants survived, and why?
* What contract does your logic guarantee, and did your tests prove it?
* How did mutation testing help you find weaknesses in your tests?
* Are there scenarios where a surviving mutant is acceptable? Why?