Testing Analysis (Part 2) – Flavia Koço

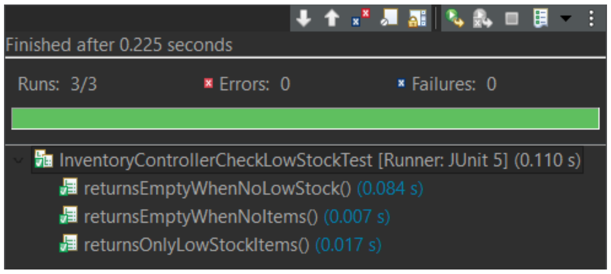
Equivalence Class Testing – InventoryController.checkLowStock()

Purpose of testing:
The purpose of this test is to verify that checkLowStock() correctly returns items that are at or below the minimum stock level, and ignores items that are above the minimum. We divide the possible stock values into equivalence classes so we don't need to test every number.

Tested Equivalence Classes & Results:

1) Low-stock class (stockQuantity ≤ minStockLevel):
Representative values tested: stock **3** with min **5**, and stock **5** with min **5** (boundary).
In both cases, the items were returned in the result list, confirming that the method correctly includes items that are below or equal to the minimum level.

2) Normal-stock class (stockQuantity > minStockLevel):
Representative values tested: stock **8** with min **5**, and stock **20** with min **5**.
The method returned an empty list (or did not include those items), confirming that items above the minimum stock are correctly excluded.


```
Finished after 0.225 seconds
Runs: 3/3        Errors: 0        Failures: 0

InventoryControllerCheckLowStockTest [Runner: JUnit 5] (0.110 s)
    returnsEmptyWhenNoLowStock() (0.084 s)
    returnsEmptyWhenNoItems() (0.007 s)
    returnsOnlyLowStockItems() (0.017 s)
```

3) Empty input class (no items saved):
Representative value tested: an empty items.dat file.
The method returned an empty list, confirming safe handling when there are no items.

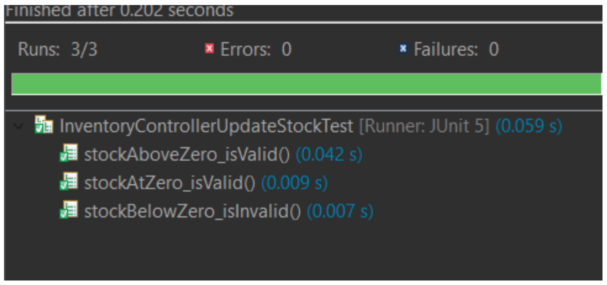
Conclusion:
Equivalence Class Testing confirmed that checkLowStock() consistently separates low-stock items from normal-stock items (including the boundary case where stock equals minimum) and safely returns an empty list when there is no data.

Boundary Value Testing – InventoryController.updateItemStock(String itemId, int quantity)

Purpose of testing:
The purpose of Boundary Value Testing is to check how updateItemStock() behaves at the most important numeric limit: the item stock must not become negative. Since the method updates stock using a number (quantity) boundary testing is suitable to confirm that stock updates are accepted at the limit and rejected just beyond it. In our tests, the item starts with **stock = 10**, so the critical boundary is when the updated stock becomes **0** (lowest valid value) and **-1** (first invalid value).

Tested boundary values & results:

1. **Stock becomes exactly 0 (boundary value):**
   We used quantity = -10, which makes the stock go from 10 to **0**.
   The method returned **true**, confirming that reaching exactly zero stock is allowed and the update is correctly applied.



```
Finished after 0.202 seconds
Runs: 3/3          Errors: 0          Failures: 0

InventoryControllerUpdateStockTest [Runner: JUnit 5] (0.059 s)
    stockAboveZero_isValid() (0.042 s)
    stockAtZero_isValid() (0.009 s)
    stockBelowZero_isInvalid() (0.007 s)
```

2. **Stock becomes -1 (just below the boundary):**
   We used quantity = -11, which makes the stock go from 10 to **-1**. The method returned **false**, confirming that updates that would make stock negative are correctly rejected.

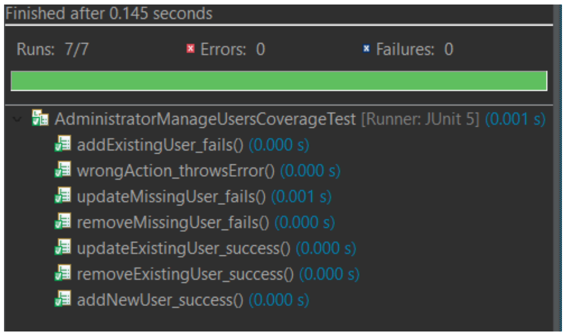
3. **Stock becomes 1 (just above the boundary):**
   We used quantity = -9, which makes the stock go from 10 to **1**.
   The method returned **true**, confirming that updates slightly above the boundary are accepted as valid.

Conclusion:
Boundary Value Testing showed that updateItemStock() correctly enforces the lower stock limit by allowing updates that keep stock at **0 or above**, and rejecting updates that would drop the stock below **0**.

Code Coverage Testing – Administrator.manageUsers(User user, String action)

Purpose of testing
The purpose of Code Coverage Testing is to verify that all important logical branches of the manageUsers method are executed and validated through test cases. This ensures that user management actions behave correctly under different conditions and that no decision paths remain untested.

Tested branches & results
The test cases cover all valid actions (add, remove, and update) for both possible conditions: when the user exists in the system and when the user does not exist. When the user does not exist, adding succeeds while removing and updating correctly fail. When the user already exists, adding fails while removing and updating succeed.
In addition an invalid action string was tested, and the method threw an IllegalArgumentException, confirming that incorrect actions are properly rejected.



```
Finished after 0.145 seconds
Runs: 7/7          Errors: 0          Failures: 0

AdministratorManageUsersCoverageTest [Runner: JUnit 5] (0.001 s)
    addExistingUser_fails() (0.000 s)
    wrongAction_throwsError() (0.000 s)
    updateMissingUser_fails() (0.001 s)
    removeMissingUser_fails() (0.000 s)
    updateExistingUser_success() (0.000 s)
    removeExistingUser_success() (0.000 s)
    addNewUser_success() (0.000 s)
```

Conclusion
The tests successfully executed all logical branches of the manageUsers method, providing full coverage of valid actions and error handling and ensuring the method behaves correctly in all situations.