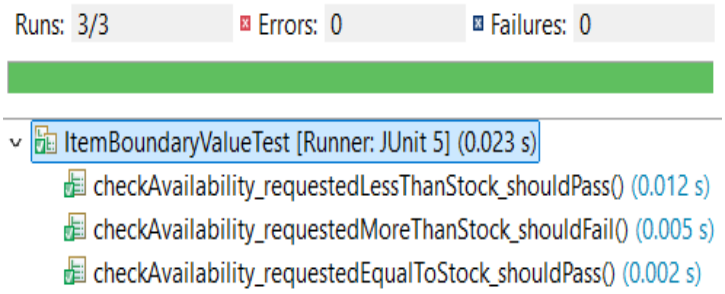
Testing Analysis (Part 2) – Jagoda Andrea

Boundary Value Testing – Item.checkAvailability(int requestedQuantity)

Purpose of testing:
To verify how the method behaves at critical boundary limits of the requested quantity in relation to the available stock, since incorrect handling of these boundary values may allow invalid purchase requests or incorrectly reject valid ones.
In conclusion, Boundary Value Testing demonstrated that the checkAvailability method correctly handles boundary conditions related to stock availability by accepting requests within the available stock limit and rejecting requests that exceed the current stock.

Tested Boundary Values & Results:
1) *requestedQuantity = 0 (zero value):*
The method returned **true**, meaning a request for zero items is considered available. This behavior is acceptable, as requesting zero quantity does not exceed the available stock and does not cause any inconsistency.



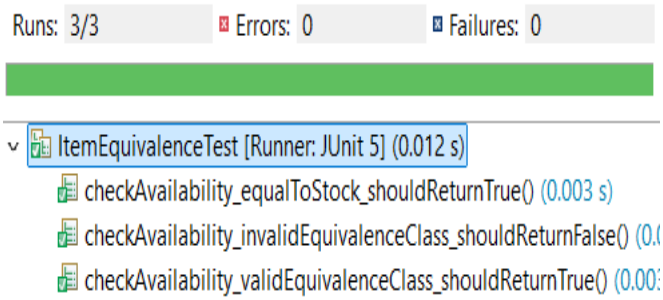
2) *requestedQuantity = 10 (equal to stock):*
The method returned **true**, confirming that the method correctly allows requests where the requested quantity is exactly equal to the available stock. This represents a valid boundary condition and shows correct behavior.
3) requestedQuantity = 11 (greater than stock):
The method returned **false**, indicating that the request exceeds the available stock. This confirms that the method correctly rejects invalid requests beyond the stock limit, preventing inconsistent inventory usage.

Equivalence Class Testing – Item.checkAvailability(int requestedQuantity)

Purpose of testing:
To verify that the checkAvailability method behaves correctly for different groups of input values by dividing the possible inputs into valid and invalid equivalence classes. This approach ensures that the method consistently accepts valid requests within stock limits and rejects requests that exceed the available stock.
In conclusion, Equivalence Class Testing confirmed that the method correctly distinguishes between valid and invalid quantity requests, providing consistent and reliable availability checks.

Tested Equivalence Classes & Results:

1) **Valid equivalence class** *(requestedQuantity ≤ stock):*



Representative values tested: 5 and 10
For *requestedQuantity = 5*, the method returned **true**, confirming that availability is correctly reported when the requested quantity is less than the available stock.
For *requestedQuantity = 10*, the method returned **true**, showing that requesting exactly the available stock is handled correctly and considered valid.

2) **Invalid equivalence class** *(requestedQuantity > stock):*
Representative value tested: 11
For *requestedQuantity = 11*, the method returned **false**, indicating that requests exceeding the available stock are correctly rejected.

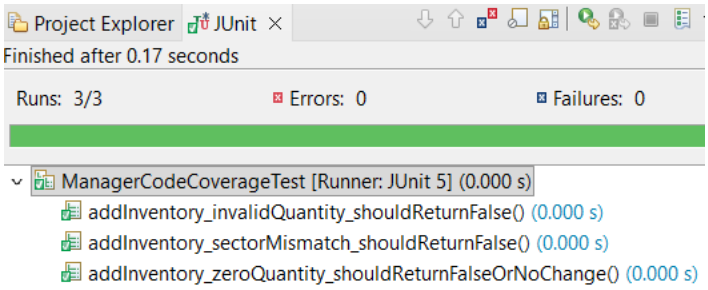
Code Coverage Testing – Manager.addInventory(Item item, int quantity)

Purpose of testing

The purpose of Code Coverage Testing is to verify that all important logical branches of the addInventory method are executed and validated through test cases. This ensures that the method behaves correctly under different conditions and that no critical decision paths remain untested.

Tested Branches & Results

1) Invalid quantity (quantity = 0):
The test case was designed to check how the method behaves when an invalid quantity is provided.
The method returned **false**, confirming that zero quantity inputs are correctly rejected and no inventory changes are applied.



2) Sector mismatch:
This test verifies the branch where the item's category sector does not match the manager's sector. The method returned **false**, indicating that inventory updates are correctly prevented when sector consistency rules are violated.

3) Invalid quantity (defensive branch):
An additional test was included to cover the logical branch that handles invalid quantity inputs. The method returned **false**, confirming that the validation logic consistently blocks improper inventory updates.

Conclusion

The executed tests provide confidence that the method's internal conditions and branches are properly implemented and protected against incorrect inventory updates.