**Q1.a. Equivalence Classes Testing**

**Testing for AccountStatus(clientAccount):**

Below is a graph of the equivalence classes to test the AccountStatus(clientAccount) function. The best-case scenario of course will be a strong test between ageFactor and balanceFactor, so 49 tests, but we can get away with weaker testing since the various internal operations are very basic. At minimum each of the accountFactors need to be tested, so that’s 5 tests, but that leaves a lot of room for error. I propose a weak test that runs 25 times, once for each bracket of the ageFactor listed in the table. Again since the various internal operations are very basic we can assume for a single-fault error if any arises. This test is also robust since it covers the invalid tests of ageFactor = 0 or balanceFactor = 0.

|  |  |  |
| --- | --- | --- |
| **ageFactor** | **balanceFactor** | **accountFactor / AccountStatus** |
| age < 18 | Any balance | accountFactor = 0 / invalid |
| age >= 95 |
| Any age | balance <= 0 |
| balance >= 5000 |
| 18 <= age < 25 | balance < 100 | 0 < accountFactor < 150 / adverse |
| 25 <= age < 35 |
| 35 <= age < 45 |
| 65 <= age < 95 |
| 45 <= age < 65 | balance < 100 | 150 <= accountFactor < 600 / acceptable |
| 18 <= age < 25 | 100 <= balance < 500 |
| 25 <= age < 35 |
| 35 <= age < 45 |
| 65 <= age < 95 |
| 18 <= age < 25 | 500 <= balance < 1000 |
| 25 <= age < 35 |
| 35 <= age < 45 |
| 45 <= age < 65 | 100 <= balance < 500 | 600 <= accountFactor < 1000 / good |
| 65 <= age < 95 | 500 <= balance < 1000 |
| 18 <= age < 25 | 1000 <= balance < 3000 |
| 25 <= age < 35 |
| 35 <= age < 45 | 1000 <= balance < 3000 | 1000 <= accountFactor / excellent |
| 45 <= age < 65 |
| 65 <= age < 95 |
| 35 <= age < 45 | 3000 <= balance < 5000 |
| 45 <= age < 65 |
| 65 <= age < 95 |

Upon further review, I can decrease the number of tests necessary from the 25 seen above down to 19 by separating out the testing:

Run a weak test on getAgeFactor(clientAccount) (7 tests), run a weak test on getBalanceFactor(clientAccount) (7 tests), and run a weak test on accountStatus(clientAccount) (5 tests). This can be done using the stubs in mocha, and will make it much easier to find any errors.

**Testing for creditStatus(clientAccount, creditCheckMode)**

This test can be done weakly, once for the invalid case and once for each class of credit score.

|  |  |  |
| --- | --- | --- |
| **creditScore** | **scoreThreshold / creditCheckMode** | **expected** |
| creditScore < 0 | Any creditCheckMode | invalid |
| 0 < creditScore < 65 | creditCheckMode = restricted | adverse |
| 65 <= creditScore | creditCheckMode = default | good |
| 0 < creditScore < 75 | creditCheckMode = restricted | adverse |
| 75 <= creditScore <= 100 | creditCheckMode = default | good |
| creditScore > 100 | Any creditCheckMode | invalid |

**Testing for productStatus(product,inventory,inventoryThreshold)**

This is very vague as we’re not given an idea of how many products are in the inventory, so I cannot recommend a strong test as it may be very taxing on the test resources if the number of unique products is large. Additionally, given the simple nature of the product and inventory variables, it can be assumed that the products are by and large equivalent in the eyes of the software. As such, a robust weak test can be done with very few runs and still give a good idea as to the software’s performance. It should be noted that one additional test should be dedicated to a product with a string that mismatches inventory name to comply with proper robust testing.

|  |
| --- |
| productQuantity (where product == inventory.productName) |
| productQuantity = 0 |
| productQuantity < inventoryThreshold |
| productQuantity = inventoryThreshold |
| productQuantity > inventoryThreshold |
| productQuantity (where product != inventory.productName) |
| “invalid” |

**Testing for orderHandling (clientAccount,product,inventory,inventoryThreshold,creditCheckMode)**

This function I recommend a strong test for, as it’s all derived from the previous functions and values. That being said, we don’t need to test for every single input, as some are already tested, or are incorporated into variables in this function. This is best seen when examining the code, all five inputs become three variables a[ccount]Status, c[redit]Status, and p[roduct]Status. These three variables correspond to the functions we test for listed above in the document, so those tests should catch any errors present. Examining these variables shows 5, 3, and 4 possible values respectively, which when run through a strong test will lead to 5\*3\*4 = 60 tests which will catch every error.

**Q1.b. Boundaries Values Testing**

Not every function needs to be boundary value tested, either due to the values being strings, or because some boundaries are tested in the equivalence class tests explained above. Boundary values can be identified by taking the number *n* on either end of the inequalities presented, and then testing for n-1, n, and n+1 for all greater/less than or equal to inequalities, and testing n-1 and n for all greater/less than inequalities.

**Testing for AccountStatus(clientAccount):**

I initially devised the boundaries to look like the table below, where the test will be run for each age value, repeating balance values where necessary, resulting in 66 tests.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ageFactor** | **Age Boundary Values** | **balanceFactor** | **Balance Boundary Values** | **accountFactor / AccountStatus** |
| age < 18 | Age = 18, 19 | Any balance | Do not care | accountFactor = 0 / invalid |
| age >= 95 | Age = 94, 95, 96 |
| Any age | Do not care | balance <= 0 | Balance = -1, 0, 1 |
| balance >= 5000 | Balance = 4999, 5000, 5001 |
| 18 <= age < 25 | age- = 17, 18, 19  age+ = 44, 45 | balance < 100 | balance = 99, 100 | 0 < accountFactor < 150 / adverse |
| 25 <= age < 35 |
| 35 <= age < 45 |
| 65 <= age < 95 | age- = 64, 65, 66  age+ = 94, 95 |
| 45 <= age < 65 | age- = 44, 45, 46  age+ = 64, 65 | balance < 100 | balance = 99, 100 | 150 <= accountFactor < 600 / acceptable |
| 18 <= age < 25 | age- = 17, 18, 19  age+ = 44, 45 | 100 <= balance < 500 | balance- = 99, 100, 101  balance+ = 499, 500 |
| 25 <= age < 35 |
| 35 <= age < 45 |
| 65 <= age < 95 | age- = 64, 65, 66  age+ = 94, 95 |
| 18 <= age < 25 | age- = 17, 18, 19  age+ = 44, 45 | 500 <= balance < 1000 | balance- = 499, 500, 501  balance+ = 999, 1000 |
| 25 <= age < 35 |
| 35 <= age < 45 |
| 45 <= age < 65 | age- = 44, 45, 46  age+ = 64, 65 | 100 <= balance < 500 | balance- = 99, 100, 101  balance+ = 499, 500 | 600 <= accountFactor < 1000 / good |
| 65 <= age < 95 | age- = 64, 65, 66  age+ = 94, 95 | 500 <= balance < 1000 | balance- = 499, 500, 501  balance+ = 999, 1000 |
| 18 <= age < 25 | age- = 17, 18, 19  age+ = 34, 35 | 1000 <= balance < 3000 | balance- = 999, 1000, 1001  balance+ = 2999, 3000 |
| 25 <= age < 35 |
| 35 <= age < 45 | age- = 34, 35, 36  age+ = 94, 95 | 1000 <= balance < 3000 | balance- = 999, 1000, 1001  balance+ = 2999, 3000 | 1000 <= accountFactor / excellent |
| 45 <= age < 65 |
| 65 <= age < 95 |
| 35 <= age < 45 | age- = 34, 35, 36  age+ = 94, 95 | 3000 <= balance < 5000 | balance- = 2999, 3000, 3001  balance+ = 4999, 5000 |
| 45 <= age < 65 |
| 65 <= age < 95 |

When I revised my equivalence classes for accountStatus however, I realised I could cut down the borders too. The borders are now:

|  |  |  |
| --- | --- | --- |
| **age** | **balance** | **accountFactor** |
| 18 | 0 | 0 |
| 25 | 100 | 150 |
| 35 | 500 | 600 |
| 45 | 1000 | 1000 |
| 65 | 3000 |  |
| 95 | 5000 |  |

**Boundary testing for creditStatus(clientAccount, creditCheckMode)**

creditScore does not have any functionality for values outside the boundaries given, i.e. [0, 100], and the boundaries of the scoreThreshold i.e. 65 and 75, so those are the borders that will be tested for, ignoring the two out of bounds tests of -1 and 101: creditScore = 0, 1, 64, 65, 66, 74, 75, 76, 99, 100

**Boundary testing for productStatus(product,inventory,inventoryThreshold)**

Since there is no functionality in the code for a negative productQuantity or a productQuantity above 1000, the border needs one set of tests at productQuantity = 0, 1, 999, 1000. Another border exists when productQuantity == inventoryThreshold, so an arbitrary inventory threshold will be selected and the test needs to be done for productQuantity = inventoryThreshold-1, inventoryThreshold, inventoryThreshold+1.

**Q1.c. Decision Table Testing**

Decision Tables are not necessary for accountStatus, creditStatus, or productStatus due to the weak tests I’m running on them. Below is the decision table for orderHandling(clientAccount, product, inventory, inventoryThreshold, creditCheckMode).

Note: in the Condition Stubs, an “X” entry indicates a “Do Not Care”, as in the variable value can be any non-“invalid” as it is not considered in the decision.

**orderHandling(clientAccount, product, inventory, inventoryThreshold, creditCheckMode)**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Condition Stubs |  | Rules | | | | | | | | | | | | | | | |
| accountStatus | excellent | good | acceptable | adverse | acceptable | acceptable | adverse | good | acceptable | invalid | X | X | acceptable | acceptable | adverse | adverse |
| creditStatus | X | good | good | good | good | good | good | adverse | adverse | X | invalid | X | adverse | adverse | good | adverse |
| productStatus | X | X | available | available | limited | soldout | limited | X | available | X | X | invalid | limited | soldout | soldout | X |
| Action Stubs | accepted | X | X | X | X |  |  |  |  |  |  |  |  |  |  |  |  |
| pending |  |  |  |  | X | X | X |  |  |  |  |  |  |  |  |  |
| underReview |  |  |  |  |  |  |  | X | X |  |  |  |  |  |  |  |
| rejected |  |  |  |  |  |  |  |  |  | X | X | X | X | X | X | X |

**Q1.d. mocha**

**i.**

See below for example test cases, see attached code test.js document for the proper coded test cases.

**Test case example for getAgeFactor**

|  |  |
| --- | --- |
| Name | TC1 – TC7 |
| Requirement | F1: getAgeFactor |
| Preconditions | Test document loaded in initial state |
| Steps | 1. Set age to somewhere in the desired age bracket being testing for |
| Expected result | ageFactor value should correspond with the age bracket being tested for |

**Test case example for getBalanceFactor**

|  |  |
| --- | --- |
| Name | TC8 – TC14 |
| Requirement | F2: getBalanceFactor |
| Preconditions | Test document loaded in initial state |
| Steps | 1. Set balance to somewhere in the desired balance bracket being testing for |
| Expected result | balanceFactor value should correspond with the balance bracket being tested for |

**Test case example for accountStatus**

|  |  |
| --- | --- |
| Name | TC15 – TC19 |
| Requirement | F3: accountStatus |
| Preconditions | Value for accountFactor is stubbed |
| Steps | 1. Stub accountFactor to somewhere in the desired bracket |
| Expected result | accountStatus string value should correspond with the accountFactor bracket being tested |

**Test case example for creditStatus**

|  |  |
| --- | --- |
| Name | TC20 – TC25 |
| Requirement | F4: creditStatus |
| Preconditions | Test document loaded in initial state |
| Steps | 1. Set creditScore to desired value 2. Set creditCheckMode to desired value |
| Expected result | creditStatus should correspond with the value of creditScore in the equivalence classes table seen above |

**Test case example for productStatus**

|  |  |
| --- | --- |
| Name | TC26 – TC30 |
| Requirement | F5: productStatus |
| Preconditions | Product string and inventory.productName string match |
| Steps | 1. Set inventory threshold to the desired value 2. Set the productQuantity to the desired value |
| Expected result | balanceFactor value should correspond with the balance bracket being tested for |

**Test case example for orderHandling**

|  |  |
| --- | --- |
| Name | TC31 – TC90 |
| Requirement | F6: orderHandling |
| Preconditions | Product string and inventory.productName string match |
| Steps | 1. Stub the accountStatus to the desired value 2. Stub the creditStatus to the desired value 3. Stub the productStatus to the desired value |
| Expected result | The value returned by orderHandling should correspond to the output values in the decision table based on the inputs stubbed |

**ii.**

Stubs were used in testing for F3: accountStatus and F6: orderHandling using the sinon library. See the code for F3 and F6 in the attached code test.js document for my use of stubs.

**iii.**

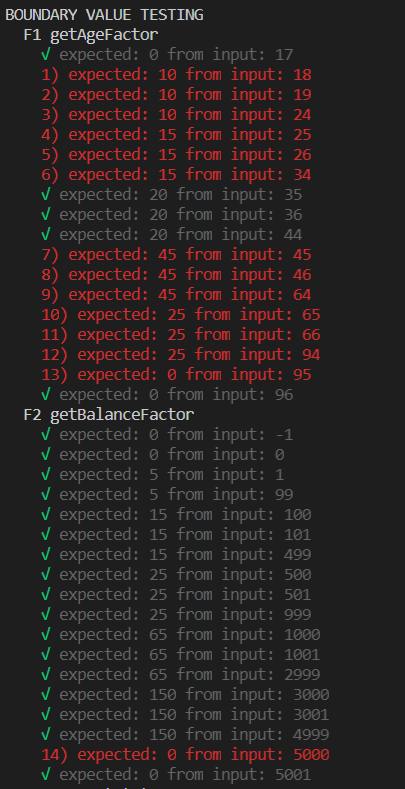
Below are the initial test results for the equivalence class testing, boundary value testing, and decision table testing. Only one bug has been fixed at this point: The syntax error on line 166 (aStatu s => aStatus) as VS Code identified it when I first saved the file.

**Initial Testing**

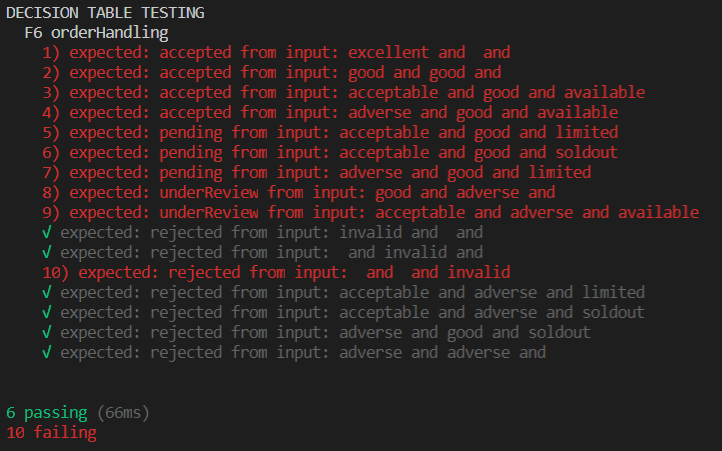
**Equivalence Class Testing**



**Boundary Value Testing**

**Decision Table Testing**



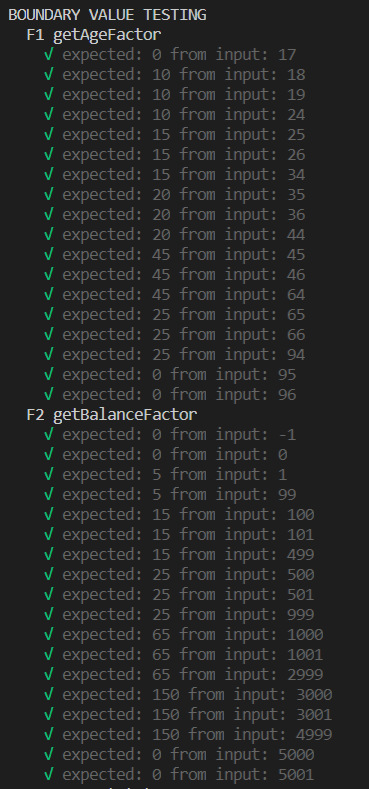
Bugs were identified and fixed in the purchaseOrderF20-fixed.js document, and comments were left where changes were made.

**After Identifying and Fixing Bugs**

**Equivalence Class Testing**



**Boundary Value Testing**

A few more bugs were identified using the boundary testing, one in line 4, one in line 24, and one in line 74

**Decision Table Testing**

