Software Testing and Validation

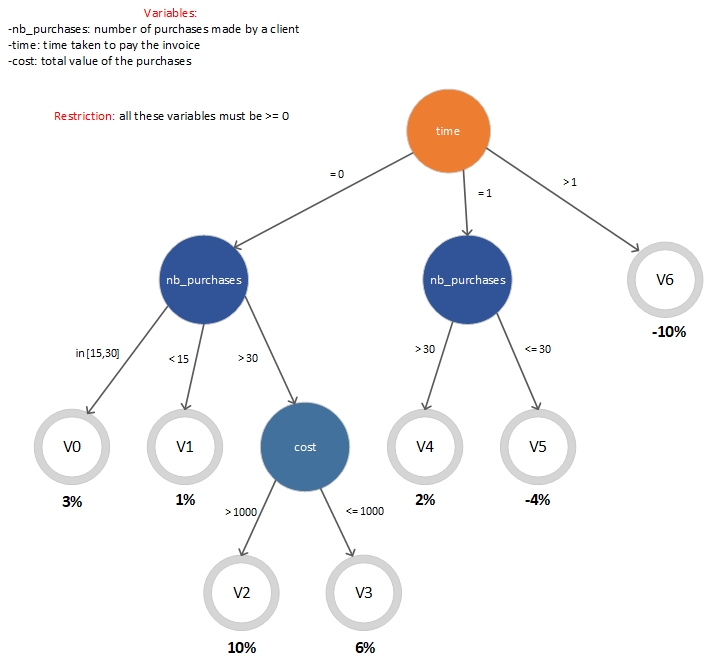
**Project Report - 2019-2020**

Group 10 - João Freitas (87671), Diogo Faustino (97081)

**Test cases for computeCreditBill method**

To test this method, we applied the Combinational Functional Test Pattern because of the complex logic behind de process of choosing the discount value.

We elaborated a decision tree:

Fig. 1 – Decision tree regarding discount value

Boundary conditions for each variant:

* V0 -> time = 0 & 15 <= nb\_purchases <= 30
* V1 -> time = 0 & nb\_purchases < 15
* V2 -> time = 0 & nb\_purchases > 30 & cost > 1000
* V3 -> time = 1 & nb\_purchases > 30 & cost <= 1000
* V4 -> time = 1 & nb\_purchases > 30
* V5 -> time = 1 & nb\_purchases <= 30
* V6 -> time > 1

**Domain matrixes for variants**















**Description of the test cases**

* In total we have 21 test cases
* We made a domain matrix for each variant in order to exercise all the branches in the graph. In the matrix, each row represents a set of input values and each column a valid or invalid combination of instance variables
* For all conditions we have one On point and one OFF point, except for the cases the condition is an equality. For those cases we have one ON point and two OFF points
* The expected results marked with an X are test cases that contain an invalid value for time (-1) which isn’t supposed to happen because the time variable must be >= 0. As it was said in the project description, the expected result for these test cases is that they throw an InvalidOperationException exception.
* The expected results marked with a variant number are test cases that belong to another variant, so we don’t need to repeat them

**Test cases for PostOffice class**

We identified the type of this class as non-modal because the constraints are not related to the history nor the message sequences. As a result, we applied the Non-modal Test Pattern.

We started by identifying the class invariant by analyzing the restrictions:

* It is impossible to have two products with the same name registered in the same post office **(for any p1,p2 in PostOffice.products, p1.name = p2.name => p1 = p2)**
* The total amount of products presented at a post office cannot exceed a given threshold(…) This maximum number of products can vary between 2 and 20 and it is specified when you create a post office **(for each PostOffice as po, po.products.size() <= po.maxNumberOfProducts & 2 <= po.maxNumberOfProducts <= 20)**
* The unit price and the number of units of a product cannot be a negative number **(for each p in PostOffice.products, p.price >=0 & p.quantity >= 0)**

**PostOffice class invariant:** for any p1,p2 in PostOffice.products, p1.name = p2.name => p1 = p2 & for each PostOffice as po, po.products.size() <= po.maxNumberOfProducts & 2 <= po.maxNumberOfProducts <= 20 & for each p in PostOffice.products, p.price >=0 & p.quantity >= 0

**Domain matrix for PostOffice class**



**Description of the test cases**

* In total we have 12 test cases
* In the matrix, each row represents a set of input values and each column a valid or invalid combination of instance variables (A - accepted, R - rejected)
* We implemented the following test cases: TC1(T,2,4,1,1), TC2(F,3,5,2,2), TC3(T,11,13,15,3), TC4(T,12,14,25,4), TC5(T,2,2,40,5) and TC6(T,5,1,45,6)

**Test cases for addNewProduct method**

To test this method we applied the Category Partition Test Pattern because, even though it depends of more than 3 variables, it only depends of 2 objects and it has a simple logic to determine if a product is added only depending of a single condition.

First, we started by understanding and identifying all functions of the addNewProduct method.

boolean **addNewProduct**(Product p) { … }

**Primary function:** Adds a new product to the post office. This method should return true under these conditions:

* if the product is not already in the post office or in other words if the product is unique in the post office`s inventory;
* if the post office has available space in the inventory;
* if the product`s price is positive
* if the unity of the product is positive

Otherwise it returns false and has no effect in the post office.

**Secondary function:** No secondary effects from this method.

Next we identified all the inputs and outputs of the MUT.

**Inputs:** Product to be added (p.name, p.price, p.currentquantity) and postOffice in cause

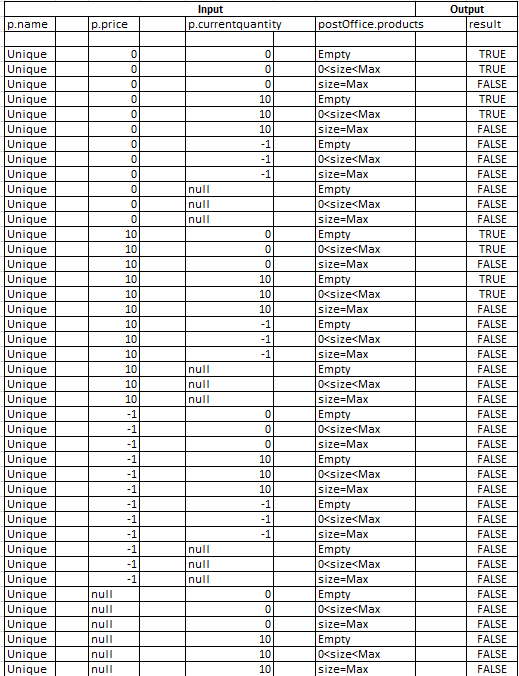
**Output:** Boolean value (result)

Then we identified categories for each input parameter and for each category we partitioned it into choices.

|  |  |  |
| --- | --- | --- |
| Variable | Category | Choice |
| p.name | Unique | p.name:= some name not in postOffice.products |
|  | Repeated | p.name:= some name in postOffice.products |
|  |  |  |
| p.price | Positive | p.price := 0, some price >= 0 |
|  | Negative | p.price := some price < 0 |
|  | Not Defined | null |
|  |  |  |
| p.currentquantity | Positive | p.currentquantity := 0, some price >= 0 |
|  | Negative | p.currentquantity := some price < 0 |
|  | Not Defined | null |
|  |  |  |
| postOffice | With space | Empty, 0<Size<Max\* |
|  | Full (size == Max\*) | Size=Max\* |
|  |  |  |

\*2<=Max<=20

Lastly, we enumerate all possible choice combinations, generate the test cases and develop the expected result for each test case.

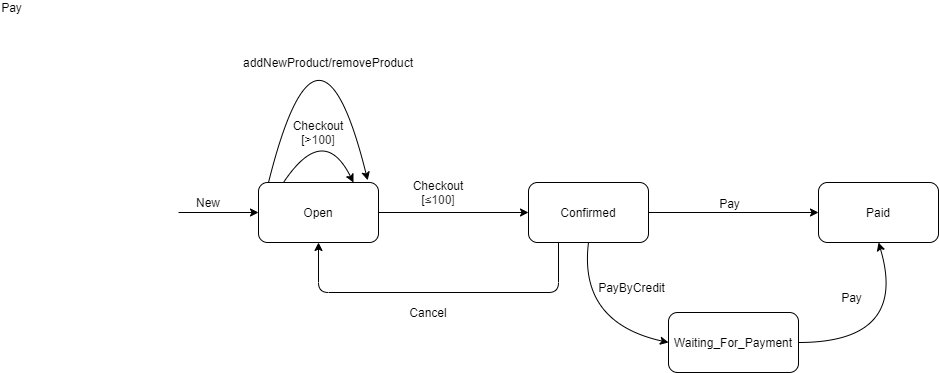


**Description of the test cases**

* In total we have 95 test cases but we only displayed 40 in the table above.
* Every parameter of product except name can be null, so it needs a Not Defined category. The postOffice can be either with space (empty or with products) or it can be already full with products (products.size is equal the defined Max for that postOffice)
* All values represented in the table above are random values that obey their respective conditions in choices and size is the number of products inside the postOffice in cause.
* The expected result for each test case indicates the output of the MUT for that possible combination.

**Test cases for Invoice class**

To test this class, we applied the FSM based testing method because the behavior of this class corresponds to a modal behavior where an invoice object has different states and transitions.

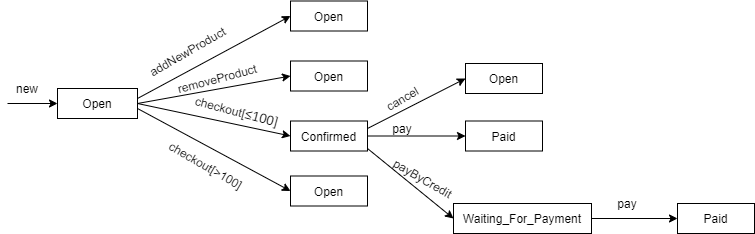
We started by designing the state machine diagram that represents the Invoice class with their respective transitions.

Then we designed a truth table for each conditional transition including any not represented in the state diagram.

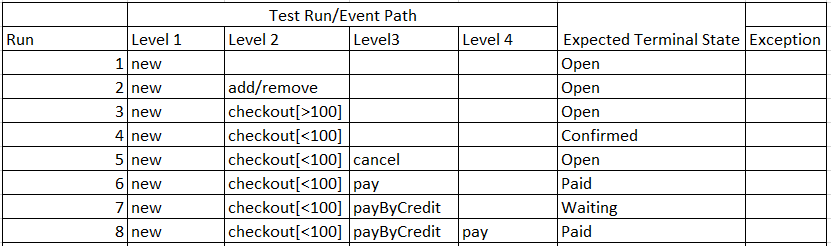
|  |  |  |  |
| --- | --- | --- | --- |
| **State** | **Message** | **Condition** | **Value** |
|  | | | |
| Open | checkout | ≤100 | Confirmed |
| Open | checkout | >100 | Open |

As we can see from the truth table and the state diagram, all conditional transitions of the CUT are already displayed in the state diagram, so we can assume that the state diagram is completed.

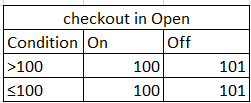
After that, we generated a transition tree based on the state diagram.



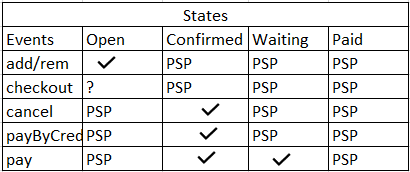
Next, we generated a conformance test suit based on the transition tree where each row it’s a different possible path.



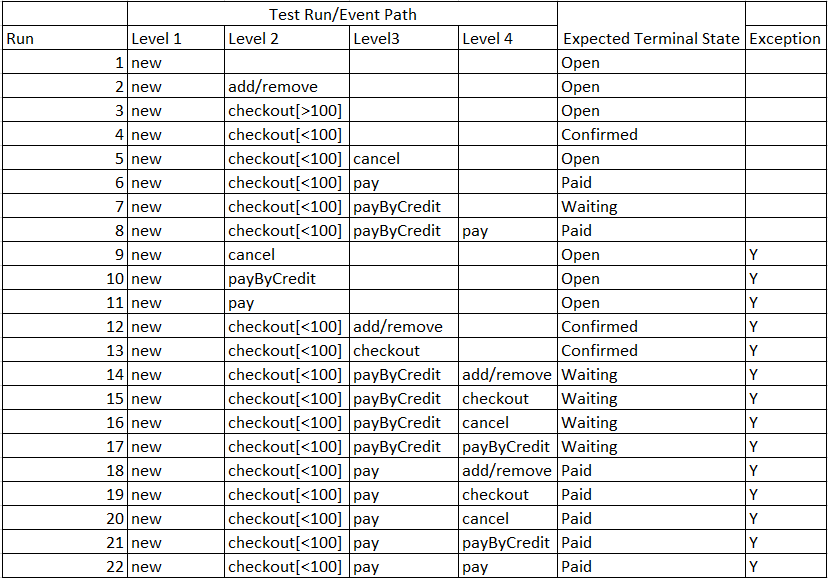
Then we developed test data for each path with a boundary condition.



Finally, we developed the Sneak Path Test Suite by building a Transition table.



And we completed the Conformance test suite and transition tree with each PSP from the previous table.



**Description of the test cases**

* In total we have 22? test cases.
* Each row in the Conformance Test Suit above represents a test case and by applying this test pattern we can test all possible transitions in this class.