Software Testing and Validation

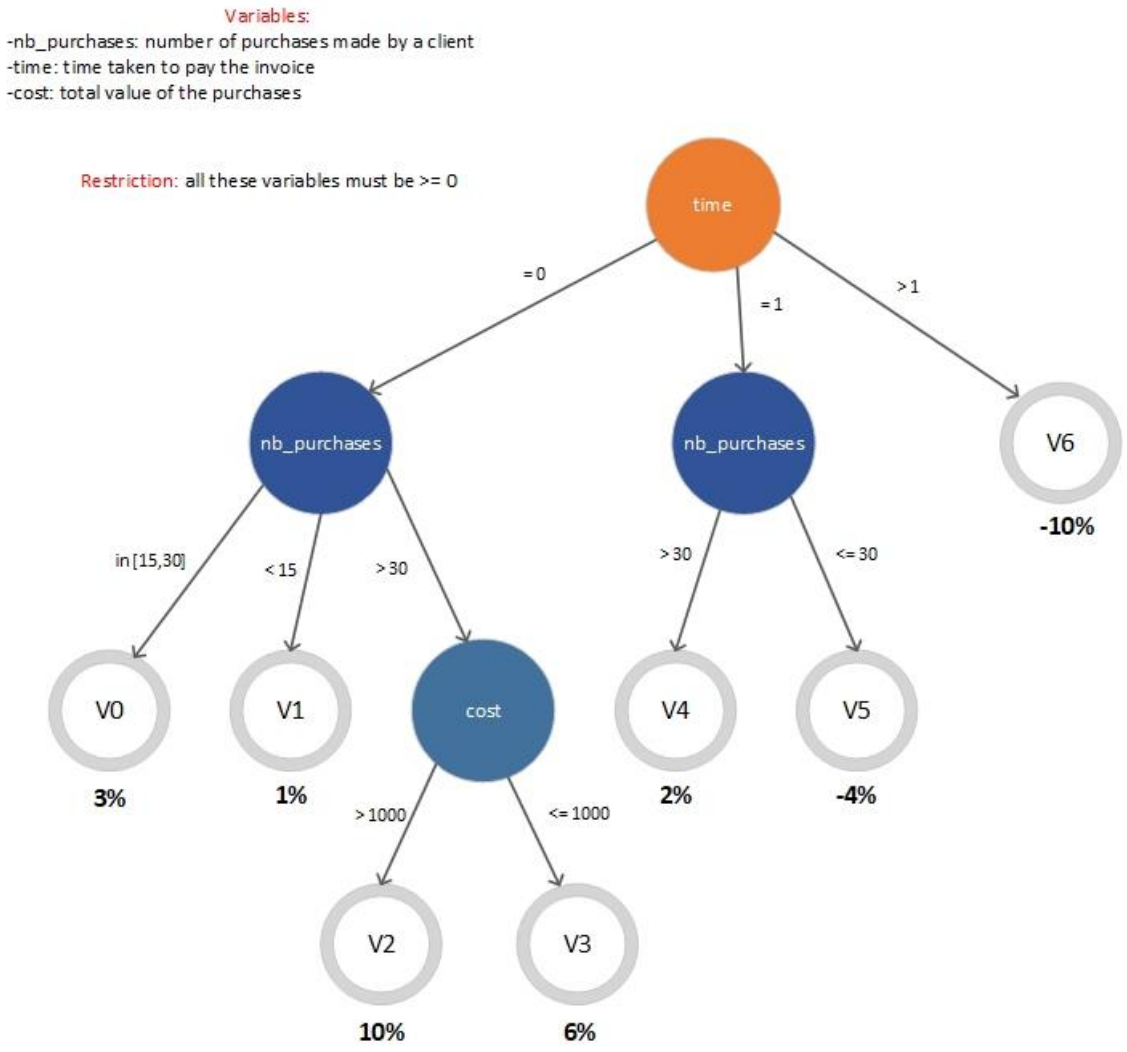
Project Report - 2019-2020

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 started by making a decision tree:



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

Then we made domain matrixes for each variant:

Domain matrixes for variants

	Variable	Condition		1	2	-	3	-	4	-
V0	time	= 0	ON	0						
			OFF		-1					
			OFF			1				
			IN				0	0	0	0
	nb_purchases	>= 15	ON				15			
			OFF					14		
		<= 30	ON						30	
			OFF							31
			IN	16	17	18				
	cost		IN	20	56	321	560	450	1005	2031
	Expected Result			**3%**	**X**	**V5**	**3%**	**V1**	**3%**	**V2**

	Variable	Condition		5	6	-	-	7	8
V1	time	= 0	ON	0					
			OFF		-1				
			OFF			1			
			IN				0	0	0
	nb_purchases	< 15	ON				15		
			OFF					14	
			IN	9	10	11			12
	cost		IN	20	50	100	500	1000	2000
	Expected Result			**1%**	**X**	**V5**	**V0**	**1%**	**1%**

	Variable	Condition		9	10	-	-	11	-	12
V2	time	= 0	ON	0						
			OFF		-1					
			OFF			1				
			IN				0	0	0	0
	nb_purchases	> 30	ON				30			
			OFF					31		
			IN	200	500	50			1000	50
	cost	> 1000	ON						1000	
			OFF							1001
			IN	1300	2500	6000	2690	8063		
	Expected Results			**10%**	**X**	**V4**	**V0**	**10%**	**V3**	**10%**

	Variable	Condition		13	14	-	-	15	16	-
V3	time	= 0	ON	0						
			OFF		-1					
			OFF			1				
			IN				0	0	0	0
	nb_purchases	> 30	ON				30			
			OFF					31		
			IN	32	33	34			40	50
	cost	<= 1000	ON						1000	
			OFF							1001
			IN	20	50	200	500	800		
	Expected Results			**6%**	**X**	**V4**	**V0**	**6%**	**6%**	**V2**

	Variable	Condition		17	-	-	-	18
V4	time	= 1	ON	1				
			OFF		0			
			OFF			2		
			IN				1	1
	nb_purchases	> 30	ON				30	
			OFF					31
			IN	40	50	80		
	cost		IN	500	100	2650	5300	3050
	Expected Result			**2%**	**V3**	**V6**	**V5**	**2%**

	Variable	Condition		19	-	-	20	-
V5	time	= 1	ON	1				
			OFF		0			
			OFF			2		
			IN				1	1
	nb_purchases	<= 30	ON				30	
			OFF					31
			IN	2	6	10		
	cost		IN	550	1023	600	3250	5064
	Expected Result			**-4%**	**V1**	**V6**	**-4%**	**V4**

	Variable	Condition		-	21
V6	time	> 1	ON	1	
			OFF		2
			IN		
	nb_purchases		IN	2	50
	cost		IN	32	500
	Expected Result			**V5**	**-10%**

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 p_1,p_2 in PostOffice.products, p_1.name = p_2.name => p_1 = p_2)**
- 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 p_1,p_2 in PostOffice.products, p_1.name = p_2.name => p_1 = p_2 & 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

Boundary			Test Cases											
variable	condition		1	2	3	4	5	6	7	8	9	10	11	12
p.name	is unique	ON	T											
		OFF		F										
	Typical	IN			T	T	T	T	T	T	T	T	T	T
po.getNumberOfProducts()	<= po.maxNumberOfProducts	ON			11									
		OFF				12								
	Typical	IN	2	3			2	5	6	7	8	9	10	11
po.maxNumberOfProducts	>= 2	ON					2							
		OFF						1						
	<= 20	ON							20					
		OFF								21				
	Typical	IN	4	5	13	14					8	12	15	17
p.price	>= 0	ON									0			
		OFF										-1		
	Typical	IN	1	2	15	25	40	45	50	78			56	89
p.quantity	>= 0	ON											0	
		OFF												-1
	Typical	IN	1	2	3	4	5	6	7	8	9	10		
Expected Results			A	R	A	A	A	R	A	R	A	R	A	R

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)

Test case implementation for PostOffice class

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)

```
package ap;

import org.testng.annotations.*;
import static org.testng.Assert.*;
import java.util.*;

@Test
public class TestPostOffice {

    /*
    * TC1
    * p.name -> is unique
    * po.getNumberOfProducts()-> 2
    * po.maxNumberOfProducts -> 4
    * p.price -> 1
    * p.quantity -> 1
    * accepted
    * */
    @Test
    public void testValidPostOfficeWithAddNewProduct(){
        PostOffice po = new PostOffice(4, new ArrayList<Product>());

        //Adds Products
        for(int i = 1; i<=2; i++){
            Product p = new Product("prod"+i,"description"+i,1,1);
            p.store(1);
            assertTrue(po.addNewProduct(p));
        }

        //Assert
        assertEquals(po.getMaxNumberOfProducts(), 4);
        assertEquals(po.getNumberOfProducts(), 2);
        List<Product> products = po.getProducts();
        for(Product p : products){
            assertEquals(p.getCurrentQuantity(), 1);
            assertEquals(p.getPrice(), 1);
        }
    }

    /*
     * TC2
     * p.name -> not unique
     * po.getNumberOfProducts()-> 3
     * po.maxNumberOfProducts -> 5
     * p.price -> 2
     * p.quantity -> 2
     * rejected
     * */
    @Test
    public void testInvalidNameforProduct(){
        List<Product> productsList = new ArrayList<Product>();
        for(int i = 1; i<=3; i++){
```

```
            Product p = new Product("prod"+i,"description"+i,2,1);
            p.store(2);
            productsList.add(p);
        }
        PostOffice po = new PostOffice(5, productsList);

        //Tries to add product with non unique name
        Product p3 = new Product("prod1","description3",2,1);
        p3.store(2);
        assertFalse(po.addNewProduct(p3));

        //Assert
        List<Product> products = po.getProducts();
        for(Product p : products){
            assertEquals(p.getCurrentQuantity(), 2);
            assertEquals(p.getPrice(), 2);
        }
        assertEquals(po.getMaxNumberOfProducts(), 5);
        assertEquals(po.getNumberOfProducts(),3);
    }

    /*
     * TC3
     * p.name -> is unique
     * po.getNumberOfProducts()-> 11
     * po.maxNumberOfProducts -> 13
     * p.price -> 15
     * p.quantity -> 3
     * accepted
     * */
    @Test
    public void testValidPostOfficeWithRemoveProductandAddNewProduct(){
        PostOffice po = new PostOffice(13, new ArrayList<Product>());
        //Adds products
        for(int i = 1; i<=12; i++){
            Product p = new Product("prod"+i,"description"+i,15,2);
            p.store(3);
            assertTrue(po.addNewProduct(p));
        }

        //Removes product to get desired quantity
        assertTrue(po.removeProduct("prod12"));

        //Assert
        assertEquals(po.getMaxNumberOfProducts(), 13);
        List<Product> products = po.getProducts();
        for(Product p : products){
            assertEquals(p.getCurrentQuantity(), 3);
            assertEquals(p.getPrice(), 15);
        }
        assertEquals(po.getNumberOfProducts(), 11);
    }

    /*
     * TC4
     * p.name -> is unique
     * po.getNumberOfProducts()-> 12
     * po.maxNumberOfProducts -> 14
     * p.price -> 25
```

```
 * p.quantity -> 4
 * accepted
 * */
@Test
public void testValidPostOfficeWithSetMaxNumberOfProducts(){
    List<Product> productsList = new ArrayList<Product>();
    for(int i = 1; i<=12; i++){
        Product p = new Product("prod"+i,"description"+i,25,3);
        p.store(4);
        productsList.add(p);
    }
    PostOffice po = new PostOffice(12, productsList);

    //Sets maxNumberOfProducts to desired value
    assertTrue(po.setMaxNumberOfProducts(14));

    //Assert
    assertEquals(po.getNumberOfProducts(), 12);
    List<Product> products = po.getProducts();
    for(Product p : products){
        assertEquals(p.getCurrentQuantity(), 4);
        assertEquals(p.getPrice(), 25);
    }
    assertEquals(po.getMaxNumberOfProducts(), 14);
}

/*
 * TC5
 * p.name -> is unique
 * po.getNumberOfProducts()-> 2
 * po.maxNumberOfProducts -> 2
 * p.price -> 40
 * p.quantity -> 5
 * accepted
 * */
@Test
public void testValidPostOfficeWithUpdate(){
    List<Product> productsList = new ArrayList<Product>();
    for(int i = 1; i<=2; i++){
        Product p = new Product("prod"+i,"description"+i,20,3);
        p.store(3);
        productsList.add(p);
    }
    PostOffice po = new PostOffice(12, productsList);

    //Updates products for desired quantity and price
    for(int i = 1; i<=2; i++){
        assertTrue(po.update("prod"+i,40,5));
    }

    //Assert
    List<Product> products = po.getProducts();
    for(Product p : products){
        assertEquals(p.getCurrentQuantity(), 5);
        assertEquals(p.getPrice(), 40);
    }
    assertEquals(po.getNumberOfProducts(), 2);
    assertEquals(po.getMaxNumberOfProducts(), 2);
}
```

```
    /*
     * TC6
     * p.name -> is unique
     * po.getNumberOfProducts()-> 5
     * po.maxNumberOfProducts -> 1
     * p.price -> 45
     * p.quantity -> 6
     * rejected
     * */
    @Test
    public void testInvalidValueforMaxNumberOfProducts(){
        List<Product> productsList = new ArrayList<Product>();
        for(int i = 1; i<=5; i++){
            Product p = new Product("prod"+i,"description"+i,45,3);
            p.store(6);
            productsList.add(p);
        }
        PostOffice po = new PostOffice(5, productsList);

        //Changes maxNumberOfProducts to invalid value
        assertFalse(po.setMaxNumberOfProducts(1));

        //Assert
        assertEquals(po.getMaxNumberOfProducts(), 5);
        assertEquals(po.getNumberOfProducts(), 5);
        List<Product> products = po.getProducts();
        for(Product p : products){
            assertEquals(p.getCurrentQuantity(), 6);
            assertEquals(p.getPrice(), 45);
        }
    }
}
```

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.

public 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;
- and if the post office has available space in the inventory;
- and if the product`s price is positive
- and 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 p 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.

Input							Output
p.name		p.price		p.currentquantity		postOffice.products	result
Unique		0		0		Empty	TRUE
Unique		0		0		0<size<Max	TRUE
Unique		0		0		size=Max	FALSE
Unique		0		10		Empty	TRUE
Unique		0		10		0<size<Max	TRUE
Unique		0		10		size=Max	FALSE
Unique		0		-1		Empty	FALSE
Unique		0		-1		0<size<Max	FALSE
Unique		0		-1		size=Max	FALSE
Unique		0		null		Empty	FALSE
Unique		0		null		0<size<Max	FALSE
Unique		0		null		size=Max	FALSE
Unique		10		0		Empty	TRUE
Unique		10		0		0<size<Max	TRUE
Unique		10		0		size=Max	FALSE
Unique		10		10		Empty	TRUE
Unique		10		10		0<size<Max	TRUE
Unique		10		10		size=Max	FALSE
Unique		10		-1		Empty	FALSE
Unique		10		-1		0<size<Max	FALSE
Unique		10		-1		size=Max	FALSE
Unique		10		null		Empty	FALSE
Unique		10		null		0<size<Max	FALSE
Unique		10		null		size=Max	FALSE
Unique		-1		0		Empty	FALSE
Unique		-1		0		0<size<Max	FALSE
Unique		-1		0		size=Max	FALSE
Unique		-1		10		Empty	FALSE
Unique		-1		10		0<size<Max	FALSE
Unique		-1		10		size=Max	FALSE
Unique		-1		-1		Empty	FALSE
Unique		-1		-1		0<size<Max	FALSE
Unique		-1		-1		size=Max	FALSE
Unique		-1		null		Empty	FALSE
Unique		-1		null		0<size<Max	FALSE
Unique		-1		null		size=Max	FALSE
Unique		null		0		Empty	FALSE
Unique		null		0		0<size<Max	FALSE
Unique		null		0		size=Max	FALSE
Unique		null		10		Empty	FALSE
Unique		null		10		0<size<Max	FALSE
Unique		null		10		size=Max	FALSE

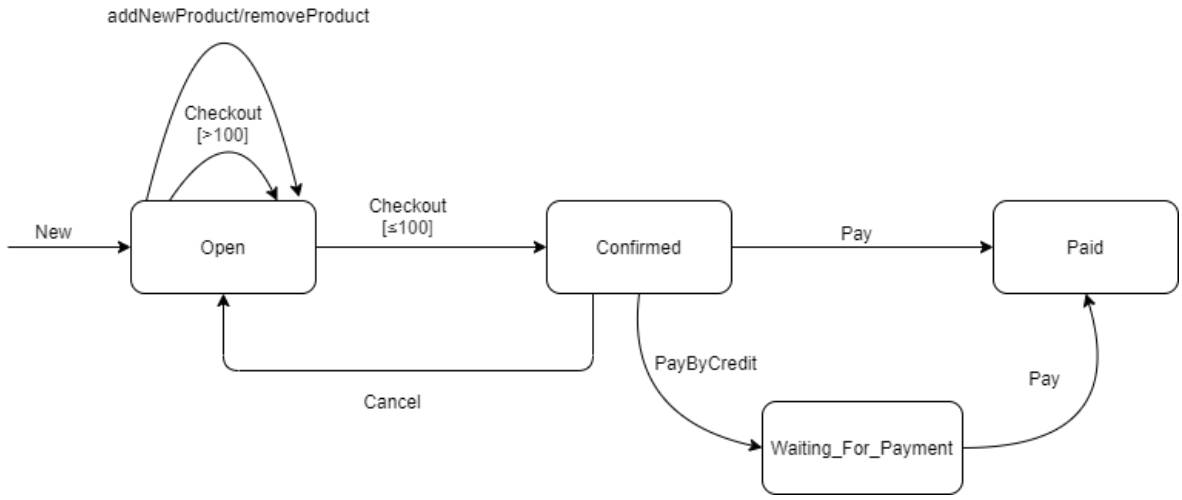
Description of the test cases

- 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 test cases and table above are random values that obey their respective conditions in choices and size is the number of products inside the postOffice in cause or in other words the products list size
- The Max value is the defined maximum of the PostOffice in cause and in the test cases it's obtained using the accessor getMaxNumberOfProducts() .
- 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 possible transitions.

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

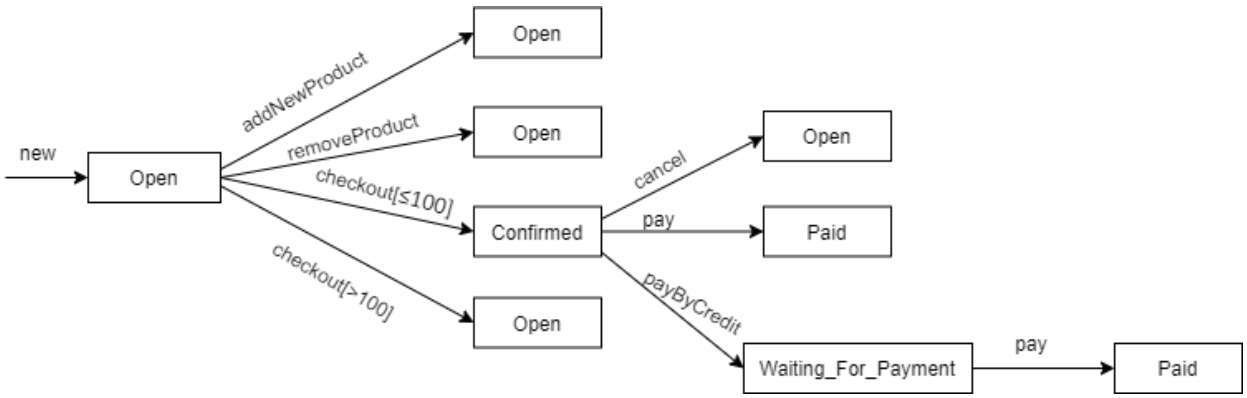


Then we designed a truth table for each conditional transition including any not represented in the state diagram.

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 an initial transition tree based on the state diagram.



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

	Test Run/Event Path					
Run	Level 1	Level 2	Level3	Level 4	Expected Terminal State	Exception
1	new				Open	
2	new	addNewProduct			Open	
3	new	removeProduct			Open	
4	new	checkout[>100]			Open	
5	new	checkout[≤100]			Confirmed	
6	new	checkout[≤100]	cancel		Open	
7	new	checkout[≤100]	pay		Paid	
8	new	checkout[≤100]	payByCredit		Waiting	
9	new	checkout[≤100]	payByCredit	pay	Paid	

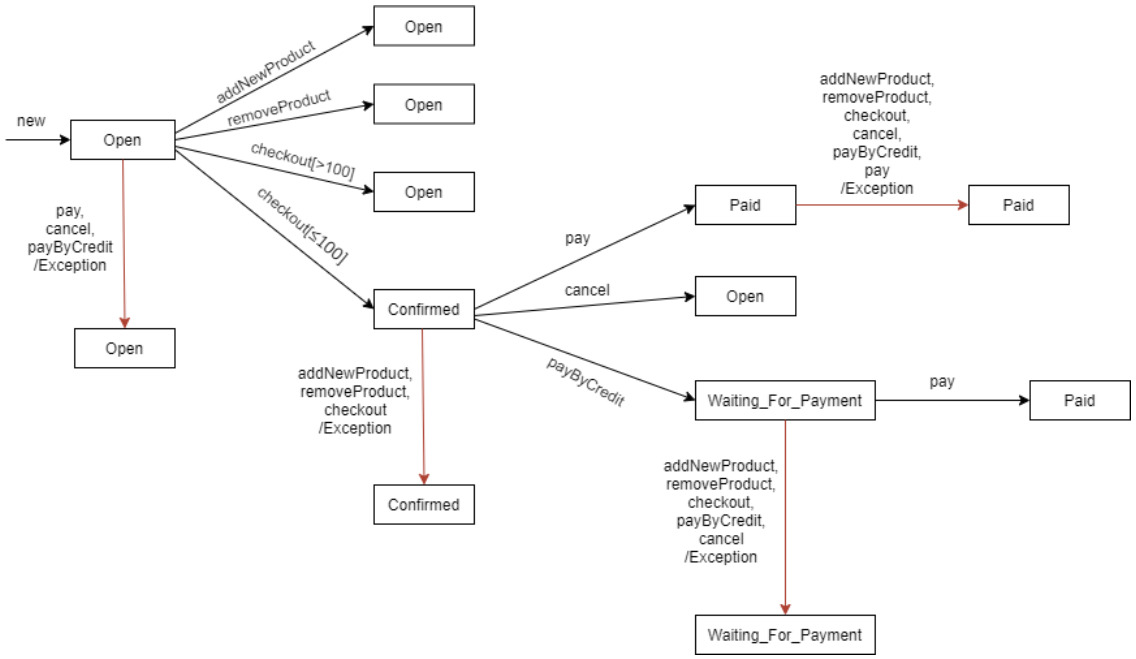
Then we developed test data for each path with a boundary condition.

checkout in Open		
Condition	On	Off
>100	100	101
≤100	100	101

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

States				
Events	Open	Confirmed	Waiting	Paid
addNewProduct	✓	PSP	PSP	PSP
removeProduct	✓	PSP	PSP	PSP
checkout	?	PSP	PSP	PSP
cancel	PSP	✓	PSP	PSP
payByCredit	PSP	✓	PSP	PSP
pay	PSP	✓	✓	PSP

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



	Test Run/Event Path					
Run	Level 1	Level 2	Level3	Level 4	Expected Terminal State	Exception
1	new				Open	
2	new	addNewProduct			Open	
3	new	removeProduct			Open	
4	new	checkout[>100]			Open	
5	new	checkout[≤100]			Confirmed	
6	new	checkout[≤100]	cancel		Open	
7	new	checkout[≤100]	pay		Paid	
8	new	checkout[≤100]	payByCredit		Waiting	
9	new	checkout[≤100]	payByCredit	pay	Paid	
10	new	cancel			Open	Y
11	new	payByCredit			Open	Y
12	new	pay			Open	Y
13	new	checkout[≤100]	addNewProduct		Confirmed	Y
14	new	checkout[≤100]	removeProduct		Confirmed	Y
15	new	checkout[≤100]	checkout		Confirmed	Y
16	new	checkout[≤100]	payByCredit	addNewProduct	Waiting	Y
17	new	checkout[≤100]	payByCredit	removeProduct	Waiting	Y
18	new	checkout[≤100]	payByCredit	checkout	Waiting	Y
19	new	checkout[≤100]	payByCredit	cancel	Waiting	Y
20	new	checkout[≤100]	payByCredit	payByCredit	Waiting	Y
21	new	checkout[≤100]	pay	addNewProduct	Paid	Y
22	new	checkout[≤100]	pay	removeProduct	Paid	Y
23	new	checkout[≤100]	pay	checkout	Paid	Y
24	new	checkout[≤100]	pay	cancel	Paid	Y
25	new	checkout[≤100]	pay	payByCredit	Paid	Y
26	new	checkout[≤100]	pay	pay	Paid	Y

Description of the test cases

- In total we have 26 test cases to test the Invoice class and in the test cases where the checkout condition is needed, it will use the test data defined above.
- Each row in the Conformance Test Suit above represents a test case and by applying this test pattern we can test all possible transitions and states from the Invoice class.
- It's expected from the first 9 test cases from the table to succeed in changing to another state and it's expected the remaining to throw an exception and remain in the same state.