

Universidade de Lisboa
Faculdade de Ciências
2016/2017



Verificação e Validação de Software
Relatório - Projecto 2

Mestrado em Informática

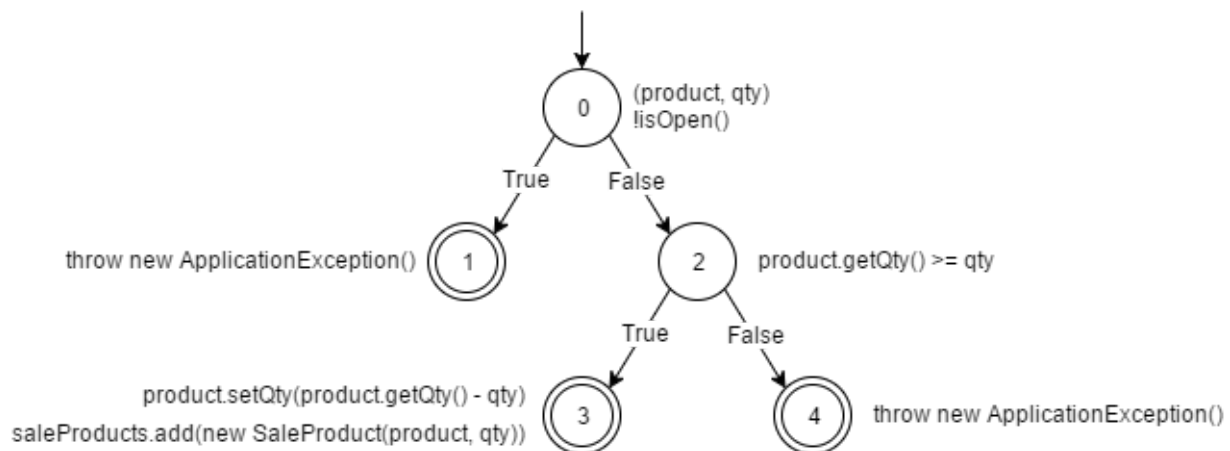
Grupo 3
Tiago Moucho, 43536
Inês de Matos, 43538

Índice

1. Testes Unitários Baseados em CFG	3
2. Testes de Mutação	5
2.1. Mutante 1	6
2.2. Mutante 2	7
2.3. Mutante 3	9
2.4. Mutante 4	10
2.5. Mutante 5	11
2.6. Mutante 6	13
2.7. Mutante 7	14
2.8. Mutante 8	15
2.9. Comparação entre Testes	16
3. Testes Unitários Baseados em Lógica	16
3.1. Cláusulas e Predicados	16
3.2. Alcance por Predicado	17
3.3. Predicados de Determinação	17
3.4. Requisitos de Teste	18
3.5. Testes Viáveis e Inviáveis	18
3.6. CACC e RACC	18
3.7. Testes JUnit	20
4. Teste de Sistema à Aplicação Web	21
5. Teste de Sistema à API REST	24

1. Testes Unitários Baseados em CFG

O método testado para ser aplicado o critério de cobertura *todos os caminhos* baseado no grafo de fluxo de controle (CFG) foi o **addProductToSale** da classe **Sale**. Para sabermos quais os caminhos a serem testados, criámos um grafo deste mesmo método, apresentado abaixo:



Todos os caminhos inferidos pelo grafo são [0, 1], [0, 2, 3] e [0, 2, 4] e os testes aplicados ao método, que cobrem os caminhos apresentados acima, encontram-se no *sale-sys-business* em */src/main/tests* na classe *SaleTestCFG.java*. Usando a ferramenta *Mockito* e fazendo *mock* à classe *Product*, os testes foram os seguintes:

```
/**
 * Test Sale when status equals CLOSED</br>
 * Path - [0,1]
 *
 * @throws ApplicationException
 */
@Test(expected = ApplicationException.class)
public void addProductToSale1() throws ApplicationException {
    Product product = mock(Product.class);
    saleTest.status = SaleStatus.CLOSED;
    when(product.getQty()).thenReturn((double) 1);
    saleTest.addProductToSale(product, 1);
}
```

No primeiro teste foi percorrido o caminho [0, 1] de forma a provocar a exceção pretendida no código (*ApplicationException.class*), em que o estado da *Sale* foi alterado para *CLOSED*.

```
/**
 * Test Sale when Product qty is less than requested</br>
 * Path - [0,2,4]
 *
 * @throws ApplicationException
 */
@Test(expected = ApplicationException.class)
public void addProductToSale2() throws ApplicationException {
    Product product = mock(Product.class);
    when(product.getQty()).thenReturn((double) 1);
    saleTest.addProductToSale(product, 2);
}
```

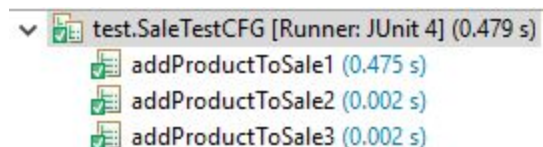
No segundo teste, o caminho percorrido foi [0, 2, 4] resultando novamente numa exceção esperada (ApplicationException), em que o stock do produto não pode ser inferior à quantidade requerida para efetuar a compra.

```
/**
 * Test Sale when Successful (must add product to saleProducts and
 * quantities much match)</br>
 * Path - [0,2,3]
 *
 * @throws ApplicationException
 */
@Test
public void addProductToSale3() throws ApplicationException {
    Product product = mock(Product.class);
    when(product.getQty()).thenReturn((double) 10);
    saleTest.addProductToSale(product, 5);

    int expectedLength = 1;
    int actualLength = saleTest.saleProducts.size();
    assertEquals(expectedLength, actualLength);

    double expectedQty = (double) 5;
    double actualQty = saleTest.saleProducts.get(0).getQty();
    assertEquals(expectedQty, actualQty, 0);
}
```

No terceiro teste, pretendemos testar o caminho [0, 2, 3], no qual efetua as operações dentro da segunda condição válida. Para tal, verificamos se o tamanho de *saleProducts* é 1 após inserir um produto e considerámos relevante reforçar o teste fazendo outra verificação com base na quantidade do produto inserido na compra.



```
test.SaleTestCFG [Runner: JUnit 4] (0.479 s)
  addProductToSale1 (0.475 s)
  addProductToSale2 (0.002 s)
  addProductToSale3 (0.002 s)
```

2. Testes de Mutação

Para cada mutante é apresentado o código que foi alterado e os testes que foram aplicados a esses mutantes. Três dos testes, **addProductToSale1**, **2** e **3**, são utilizados em vários dos testes dos mutantes por serem testes base, pois estes foram inferidos no código original. Também é mostrado o resultado em JUnit de cada teste, representando se o teste foi ou não bem sucedido.

2.1. Mutante 1

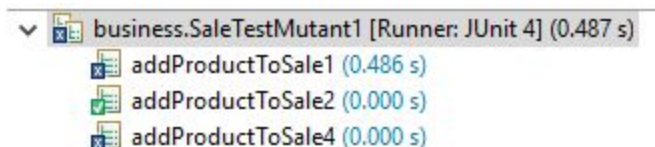
2.1.1. Operador e Mutação

```
public void addProductToSaleMutant1(Product product, double qty)
    throws ApplicationException {
    if (isOpen())
        throw new ApplicationException("Cannot add products to a closed sale.");

    // if there is enough stock
    if (product.getQty() >= qty) {
        // adds product to sale
        product.setQty(product.getQty() - qty);
        saleProducts.add(new SaleProduct(product, qty));
    } else
        throw new ApplicationException("Product " + product.getProdCod() + " has stock (" +
            product.getQty() + ") which is insufficient for the current sale");
}
```

Operador usado: UOD - Unary Operator Deletion

Mutante: if(isOpen())



```
business.SaleTestMutant1 [Runner: JUnit 4] (0.487 s)
  addProductToSale1 (0.486 s)
  addProductToSale2 (0.000 s)
  addProductToSale4 (0.000 s)
```

2.1.2. Teste que não atinja a mutação

Não é possível conceber um teste que não atinja a mutação porque o primeiro mutante afecta a primeira condição (e operação) do método, passando sempre por esta.

2.1.3. Teste que mata o mutante

Com base nos testes já feitos no ponto 1, é possível encontrar pelo menos um teste que mata o mutante, o **addProductToSale1**. Pelos requisitos *reachability*, *infection* e *propagation*, houve falha logo no primeiro teste em que é suposto dar excepção (ApplicationException) caso a *sale* esteja fechada, no entanto o esperado não acontece, prosseguindo com o resto do método.

2.1.4. Teste que mata o mutante na forma fraca

Para este teste, foi definido o estado da *sale* para *CLOSED*, passando pelo mutante sem criar exceção. Foi utilizada uma quantidade de produto válida para ser possível entrar na segunda condição e adicionar o produto à *sale*. O resultado esperado seria a adição do produto não ser válida pois a *sale* está fechada, contudo o produto é adicionado à lista de produtos, falhando assim o teste.

```
@Test
public void addProductToSale4() throws ApplicationException {
    Product product = mock(Product.class);
    saleTest.status = SaleStatus.CLOSED;
    when(product.getQty()).thenReturn((double) 2);
    saleTest.addProductToSaleMutant1(product, 2);

    int expectedSize = 0;
    int actualSize= saleTest.saleProducts.size();

    assertEquals(expectedSize, actualSize);
}
```

2.1.5. Teste que atinja a mutação e não a um estado de erro

Com bases nos testes efetuados anteriormente, existe pelo menos um teste que atinge a mutação e não leva a um estado de erro, o **addProductToSale2**. Neste teste seria expectável lançar a exceção dentro do *else*, no entanto como o *status* da *sale* está definida a *OPEN* e a exceção é a mesma, não existe nenhum erro, podendo criar um falso positivo.

2.2. Mutante 2

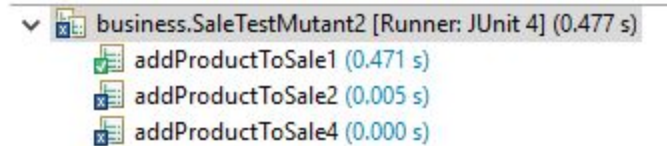
2.2.1. Operador e Mutação

```
public void addProductToSaleMutant2(Product product, double qty)
    throws ApplicationException {
    if (!isOpen())
        throw new ApplicationException("Cannot add products to a closed sale.");

    // if there is enough stock
    if (product.getQty() < qty) {
        // adds product to sale
        product.setQty(product.getQty() - qty);
        saleProducts.add(new SaleProduct(product, qty));
    } else
        throw new ApplicationException("Product " + product.getProdCod() + " has stock (" +
            product.getQty() + ") which is insufficient for the current sale");
}
```

Operador usado: ROR - Relational Operator Replacement

Mutante: `if(product.getQty() <= qty)`



2.2.2. Teste que não atinja a mutação

Com base nos testes feitos anteriormente, esta situação é alcançada pelo teste **addProductToSale1**. O estado da *sale* encontra-se *CLOSED* logo lança uma exceção sem passar pelo mutante.

2.2.3. Teste que mata o mutante

Com base nos testes feitos anteriormente, esta situação é alcançada pelo teste **addProductToSale2**. Seria expectável uma exceção, porque a quantidade de *stock* não pode ser inferior à quantidade de produtos, este mutante infecta a segunda condição, no qual propaga para resto o código, visto que este é executado indevidamente.

2.2.4. Teste que mata o mutante na forma fraca

```
@Test
public void addProductToSale4() throws ApplicationException {
    Product product = mock(Product.class);
    when(product.getQty()).thenReturn((double) 0);
    saleTest.addProductToSaleMutant2(product, 1);

    assertTrue("Expected a positive product quantity", product.getQty() - 1 > 0);
}
```

Neste caso foi necessário desenhar outro teste, o **addProductToSale4**. É testado se um produto é adicionado independentemente do tamanho. Dado que o mutante inverte a operação, a nova quantidade será negativa, o que leva a um erro, pois uma quantidade não pode ser negativa.

2.2.5. Teste que atinja a mutação e não a um estado de erro

Não existe nenhum teste que chegue à mutação e que não dê erro. Visto que, sempre que ele entra pela segunda condição, a nova quantidade do produto será sempre negativa, ou seja, um erro.

2.3. Mutante 3

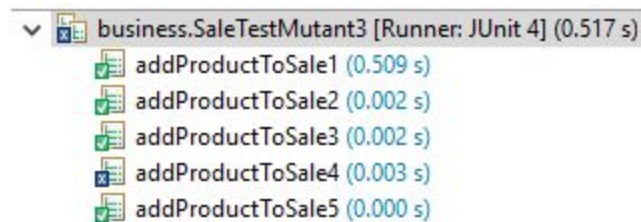
2.3.1. Operador e Mutação

```
public void addProductToSaleMutant3(Product product, double qty)
    throws ApplicationException {
    if (!isOpen())
        throw new ApplicationException("Cannot add products to a closed sale.");

    // if there is enough stock
    if (product.getQty() >= qty) {
        // adds product to sale
        product.setQty(product.getQty() + qty);
        saleProducts.add(new SaleProduct(product, qty));
    } else
        throw new ApplicationException("Product " + product.getProdCod() + " has stock (" +
            product.getQty() + ") which is insufficient for the current sale");
}
```

Operador usado: AOR - Arithmetic Operator Replacement

Mutante: product.setQty(product.getQty() + qty)



2.3.2. Teste que não atinja a mutação

Com base nos testes feitos anteriormente, esta situação é alcançada pelo teste **addProductToSale1**. O estado da *sale* encontra-se *CLOSED* logo lança uma exceção sem passar pelo mutante.

2.3.3. Teste que mata o mutante

Não foi encontrado nenhum teste que mata o mutante seguindo as três condições. Todas as ocorrências são apenas na forma fraca, ocorrendo apenas infecção sem chegar a propagação.

2.3.4. Teste que mata o mutante na forma fraca

```
@Test
public void addProductToSale4() throws ApplicationException {
    Product product = mock(Product.class);
    when(product.getQty()).thenReturn((double) 10);
    saleTest.addProductToSaleMutant3(product, 5);

    double expectedQty = (double) 5;
    double actualQty = saleTest.saleProducts.get(0).getProduct().getQty() + saleTest.saleProducts.get(0).getQty();
    assertEquals(expectedQty, actualQty, 0);
}
```


Neste caso foi necessário desenhar outro teste, o **addProductToSale4**. É testado se a quantidade de *stock* está a ser bem atualizada. Dando estando disponível 10 em *stock* e sendo pedidos 5 produtos, seria expectável que ao adicionar a *sale*, o *stock* disponível passasse a ser 5. Tal não acontece, porque o mutante executa a operação errada, o que leva a um erro na quantidade de *stock* disponível.

2.3.5. Teste que antiga a mutação e não a um estado de erro

```
@Test
public void addProductToSale5() throws ApplicationException {
    Product product = mock(Product.class);
    when(product.getQty()).thenReturn((double) 10);
    saleTest.addProductToSaleMutant3(product, 0);

    double expectedQty = (double) 10;
    double actualQty = product.getQty() + 0;
    assertEquals(expectedQty, actualQty, 0);
}
```

Neste caso foi necessário desenhar outro teste, o **addProductToSale5**. É testado se a quantidade dos produtos está a ser bem atualizada. Sempre que a quantidade de stock é 0, este passa sempre pelo mutante no entanto nunca gera um erro, pelo que o resultado vai ser sempre igual à quantidade do produto.

2.4. Mutante 4

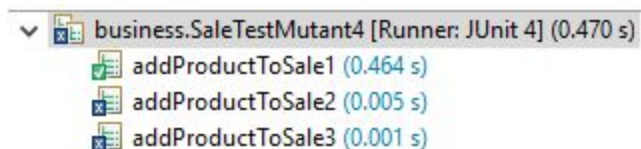
2.4.1. Operador e Mutação

```
public void addProductToSaleMutant4(Product product, double qty)
    throws ApplicationException {
    if (!isOpen())
        throw new ApplicationException("Cannot add products to a closed sale.");

    qty = qty - 1.0;
    // if there is enough stock
    if ( product.getQty() >= qty ) {
        // adds product to sale
        product.setQty(product.getQty() - qty);
        saleProducts.add(new SaleProduct(product, qty));
    } else
        throw new ApplicationException("Product " + product.getProdCod() + " has stock (" +
            product.getQty() + ") which is insufficient for the current sale");
}
```

Operador usado: SVR - Scalar Variable Replacement

Mutante: `qty = qty - 1.0;`



2.4.2. Teste que não atinja a mutação

Com base nos testes feitos anteriormente, esta situação é alcançada pelo teste **addProductToSale1**. O estado da *sale* encontra-se *CLOSED* logo lança uma exceção sem passar pelo mutante.

2.4.3. Teste que mata o mutante

Com base nos testes feitos anteriormente, esta situação é alcançada pelo teste **addProductToSale2**. Isto acontece porque seria expectável que se lançasse uma exceção, o que através do mutante tal não acontece, a variável *qty* ao ser incrementada infecta a condição *if* propagando para dentro desse bloco e executando as operações, indevidamente. No teste sem mutante este teste passaria, pois era esperado uma exceção pelo que $1 \geq 2$ é falso.

2.4.4. Teste que mata o mutante na forma fraca

Com base nos testes feitos anteriormente, esta situação é alcançada pelo teste **addProductToSale3**. Neste caso, quando o mutante é executado infecta a condição *if*, executa o que está lá dentro e gera um erro na quantidade do produto (*qty*), em vez de ser 5.0 é 4.0.

2.4.5. Teste que atinja a mutação e não a um estado de erro

Não é possível concretizar um teste que atinja esta condição, porque em todos os casos que o mutante é *reachable* vai entrar sempre em estado de erro.

2.5. Mutante 5

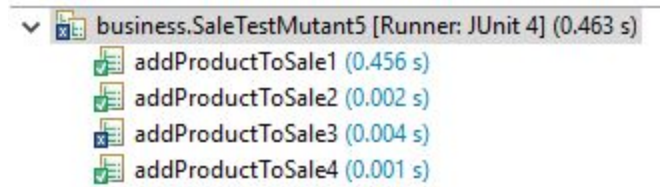
2.5.1. Operador e Mutação

```
public void addProductToSaleMutant5(Product product, double qty)
    throws ApplicationException {
    if (!isOpen())
        throw new ApplicationException("Cannot add products to a closed sale.");

    // if there is enough stock
    if (product.getQty() >= qty) {
        // adds product to sale
        product.setQty(product.getQty() - qty);
        saleProducts.add(new SaleProduct(product, product.getQty()));
    } else
        throw new ApplicationException("Product " + product.getProdCod() + " has stock (" +
            product.getQty() + ") which is insufficient for the current sale");
}
```

Operador usado: SVR - Scalar Variable Replacement

Mutante: saleProducts.add(new SaleProduct(product,product.getQty()));



2.5.2. Teste que não atinja a mutação

Com base nos testes feitos anteriormente, esta situação é alcançada pelo teste **addProductToSale1**. O estado da *sale* encontra-se *CLOSED* logo lança uma exceção sem passar pelo mutante.

2.5.3. Teste que mata o mutante

Não foi encontrado nenhum teste que mata o mutante seguindo as três condições. Todas as ocorrências são apenas na forma fraca, ocorrendo apenas infecção sem chegar a propagação.

2.5.4. Teste que mata o mutante na forma fraca

Com base nos testes feitos anteriormente, esta situação é alcançada pelo teste **addProductToSale3**. É introduzido um conjunto de quantidades válidas, no entanto é esperado que a quantidade de produtos pedidos seja igual à original e o resultado vem diferente. No entanto, não leva a falha porque não existe nenhum *output* visível.

2.5.5. Teste que atinja a mutação e não a um estado de erro

```
@Test
public void addProductToSale4() throws ApplicationException {
    Product product = mock(Product.class);
    when(product.getQty()).thenReturn((double) 5);
    saleTest.addProductToSaleMutant5(product, 5);

    double expectedQty = (double) 5;
    double actualQty = saleTest.saleProducts.get(0).getQty();
    assertEquals(expectedQty, actualQty, 0);
}
```

Neste caso foi necessário desenhar outro teste, **addProductToSale4**. Foi encontrada uma ocorrência onde satisfaz esta condição. Caso a quantidade de *stock* do produto for igual à quantidade pedida, então não irá haver nenhum erro pois o valor é igual.

2.6. Mutante 6

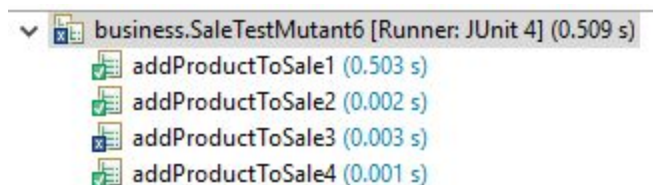
2.6.1. Operador e Mutação

```
public void addProductToSaleMutant6(Product product, double qty)
    throws ApplicationException {
    if (!isOpen())
        throw new ApplicationException("Cannot add products to a closed sale.");

    // if there is enough stock
    if (product.getQty() == qty) {
        // adds product to sale
        product.setQty(product.getQty() - qty);
        saleProducts.add(new SaleProduct(product, qty));
    } else
        throw new ApplicationException("Product " + product.getProdCod() + " has stock (" +
            product.getQty() + ") which is insufficient for the current sale");
}
```

Operador usado: ROR - Relational Operator Replacement

Mutante: `if(product.getQty() == qty)`



2.6.2. Teste que não atinja a mutação

Com base nos testes feitos anteriormente, esta situação é alcançada pelo teste **addProductToSale1**. O estado da *sale* encontra-se *CLOSED* logo lança uma exceção sem passar pelo mutante.

2.6.3. Teste que mata o mutante

Neste caso foi necessário alterar o teste **addProductToSale3**. Este teste à priori dava um erro no Junit, no entanto considerámos o teste relevante para este mutante. Assim, introduzimos apenas um *try/catch* a evitar o erro. Este teste mata o mutante porque, quando são introduzidas duas quantidades supostamente válidas, isto é em que a quantidade de *stock* é superior à quantidade de produtos pedidos, este não deveria dar exceção. Contudo, neste mutante leva a propagação de uma exceção que não deveria dar.

2.6.4. Teste que mata o mutante na forma fraca

Não existe nenhuma ocorrência em que o mutante fosse morto e que não originasse uma falha. Porque sempre que as quantidades são diferentes este mutante leva sempre a exceção(seja o *stock* menor ou maior).

2.6.5. Teste que atinja a mutação e não a um estado de erro

```
@Test(expected = ApplicationException.class)
public void addProductToSale4() throws ApplicationException {
    Product product = mock(Product.class);
    when(product.getQty()).thenReturn((double) 5);
    saleTest.addProductToSaleMutant6(product, 10);

    double expectedQty = (double) 5;
    double actualQty = saleTest.saleProducts.get(0).getQty();
    assertEquals(expectedQty, actualQty, 0);
}
```

Neste caso foi necessário desenhar outro teste, **addProductToSale4**. Este teste recebe um valor em que a quantidade de *stock* é inferior ao número de produtos pedidos, que tanto no mutante como no código original é suposto lançar, e lança, excepção.

2.7. Mutante 7

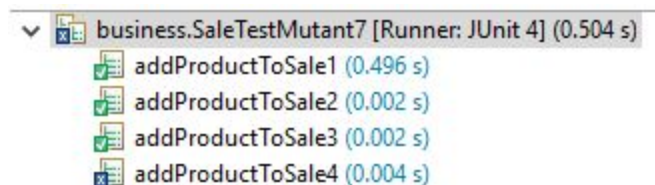
2.7.1. Operador e Mutação

```
public void addProductToSaleMutant7(Product product, double qty)
    throws ApplicationException {
    if (!isOpen())
        throw new ApplicationException("Cannot add products to a closed sale.");

    // if there is enough stock
    if (product.getQty() >= qty) {
        // adds product to sale
        product.setQty(product.getQty() - qty);
        saleProducts.add(new SaleProduct(null, qty));
    } else
        throw new ApplicationException("Product " + product.getProdCod() + " has stock (" +
            product.getQty() + ") which is insufficient for the current sale");
}
```

Operador usado: ROR - Relational Operator Replacement

Mutante: product.setQty(product.getQty() - qty)



2.7.2. Teste que não atinja a mutação

Com base nos testes feitos anteriormente, esta situação é alcançada pelo teste **addProductToSale1**. O estado da *sale* encontra-se *CLOSED* logo lança uma excepção sem passar pelo mutante.

2.7.3. Teste que mata o mutante

Não foi encontrado nenhum teste que mata o mutante seguindo as três condições. Todas as ocorrências são apenas na forma fraca, ocorrendo apenas infecção sem chegar a propagação.

2.7.4. Teste que mata o mutante na forma fraca

```
@Test
public void addProductToSale4() throws ApplicationException {
    Product product = mock(Product.class);
    when(product.getQty()).thenReturn((double) 10);
    saleTest.addProductToSaleMutant7(product, 5);

    Product newp = saleTest.saleProducts.get(0).getProduct();

    assertNotNull(newp);
}
```

Neste caso foi necessário desenhar outro teste, **addProductToSale4**. Fizemos este teste com o pressuposto que não é possível comprar um produto que não exista, ou seja que é *null*.

2.7.5. Teste que atinja a mutação e não a um estado de erro

Não é possível concretizar um teste que atinja esta condição, porque em todos os casos que o mutante é *reachable* vai entrar sempre em estado de erro.

2.8. Mutante 8

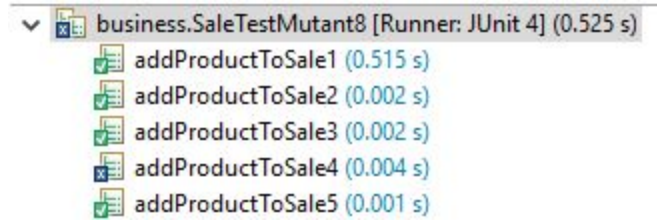
2.8.1. Operador e Mutação

```
public void addProductToSaleMutant8(Product product, double qty)
    throws ApplicationException {
    if (!isOpen())
        throw new ApplicationException("Cannot add products to a closed sale.");

    // if there is enough stock
    if (product.getQty() >= qty) {
        // adds product to sale
        product.setQty(product.getQty());
        saleProducts.add(new SaleProduct(product, qty));
    } else
        throw new ApplicationException("Product " + product.getProdCod() + " has stock (" +
            product.getQty() + ") which is insufficient for the current sale");
}
```

Operador usado:SVR - Scalar Variable Replacement

Mutante: product.setQty(product.getQty() - 0) <=> qty = 0



2.8.2. Teste que não atinja a mutação

Com base nos testes feitos anteriormente, esta situação é alcançada pelo teste **addProductToSale1**. O estado da *sale* encontra-se *CLOSED* logo lança uma exceção sem passar pelo mutante.

2.8.3. Teste que mata o mutante

Não foi encontrado nenhum teste que mata o mutante seguindo as três condições. Todas as ocorrências são apenas na forma fraca, ocorrendo apenas infecção sem chegar a propagação.

2.8.4. Teste que mata o mutante na forma fraca

```
@Test
public void addProductToSale4() throws ApplicationException {
    Product product = mock(Product.class);
    when(product.getQty()).thenReturn((double) 10);
    saleTest.addProductToSaleMutant8(product, 5);

    double expectedQty = (double) 5;
    double actualQty = saleTest.saleProducts.get(0).getProduct().getQty();
    assertEquals(expectedQty, actualQty, 0);
}
```

Neste caso foi necessário desenhar outro teste, **addProductToSale4**. É esperada uma subtração entre as duas variáveis, mas isso não acontece, infectando a quantidade de *stock* mal atualizada.

2.8.5. Teste que atinja a mutação e não a um estado de erro

```
@Test
public void addProductToSale5() throws ApplicationException {
    Product product = mock(Product.class);
    when(product.getQty()).thenReturn((double) 0);
    saleTest.addProductToSaleMutant8(product, 0);

    double expectedQty = (double) 0;
    double actualQty = saleTest.saleProducts.get(0).getProduct().getQty();
    assertEquals(expectedQty, actualQty, 0);
}
```


Neste caso foi necessário desenhar outro teste, **addProductToSale5**. Visto que $qty = 0$, para chegar a esta condição só é necessário que a quantidade em stock seja, igual, ou seja 0 (zero). Desta forma nunca será um erro, pois a variável terá o valor que seria suposto.

2.9. Comparação entre Testes

Para todos os mutantes, os três primeiros testes são comuns à cobertura de todos os caminhos e cobertura por mutação. Considerámos relevante usá-los, visto que representam os caminhos possíveis funcionais do método a ser testado. No entanto, houve necessidade de criar testes específicos para alguns mutantes, pois os três primeiros testes nem sempre cobriam os requisitos necessários para: matar; matar fracamente; não atingir; atingir e não dar erro.

3. Testes Unitários Baseados em Lógica

3.1. Cláusulas e Predicados

Foi criada uma tabela em que apresentamos as cláusulas e predicados presentes no método *isValidVat* e simplificamos as cláusulas com letras.

#	Predicado	Cláusula	
1	$(vat < 100000000 \parallel vat > 999999999)$	$vat < 100000000$	$vat > 999999999$
2	$(firstDigit \neq 1 \ \&\& \ firstDigit \neq 2)$	$firstDigit \neq 1$	$firstDigit \neq 2$
3	$(i < 10 \ \&\& \ vat \neq 0)$	$i < 10$	$vat \neq 0$
4	$checkDigitCalc == 10$	$checkDigitCalc == 10$	
5	$checkDigit == checkDigitCalc$	$checkDigit == checkDigitCalc$	

Tabela 1. Tabela de Predicados e Cláusulas

p1 = a b				p2 = c && d				p3 = e && f				p4 = g		p5 = h	
Test	a	b	a b	Test	c	d	c && d	Test	e	f	e && f	Test	g	Test	h
1	T	T	T	1	T	T	T	1	T	T	T	1	T	1	T
2	T	F	T	2	T	F	F	2	T	F	F	2	F	2	F
3	F	T	T	3	F	T	F	3	F	T	F				
4	F	F	F	4	F	F	F	4	F	F	F				

3.2. Alcance por Predicado

p	r(p)
p1	TRUE
p2	$r(p1) \ \&\& \ !p1$
p3	$r(p2) \ \&\& \ !p2$
p4	$r(p2) \ \&\& \ !p2$
p5	$r(p2) \ \&\& \ !p2$

3.3. Predicados de Determinação

P1 = a || b

$$d(a) = p[\text{true}/a] \text{ XOR } p[\text{false}/a] = (\text{true} \ || \ b) \text{ XOR } (\text{false} \ || \ b) = \text{true} \text{ XOR } b = \sim b$$

$$d(b) = p[\text{true}/b] \text{ XOR } p[\text{false}/b] = (a \ || \ \text{true}) \text{ XOR } (a \ || \ \text{false}) = a \text{ XOR } \text{true} = \sim a$$

P2 = c && d

$$d(c) = p[\text{true}/c] \text{ XOR } p[\text{false}/c] = (\text{true} \ \&\& \ d) \text{ XOR } (\text{false} \ \&\& \ d) = \text{false} \text{ XOR } d = d$$

$$d(d) = p[\text{true}/d] \text{ XOR } p[\text{false}/d] = (c \ \&\& \ \text{true}) \text{ XOR } (c \ \&\& \ \text{false}) = c \text{ XOR } \text{false} = c$$

P3 = e && f

$$d(e) = p[\text{true}/e] \text{ XOR } p[\text{false}/e] = (\text{true} \ \&\& \ f) \text{ XOR } (\text{false} \ \&\& \ f) = \text{false} \text{ XOR } f = e$$

$$d(f) = p[\text{true}/f] \text{ XOR } p[\text{false}/f] = (e \ \&\& \ \text{true}) \text{ XOR } (e \ \&\& \ \text{false}) = e \text{ XOR } \text{false} = e$$

P4 = g

$$d(g) = g \text{ xor } \text{false} \text{ xor } g$$

$$d(g) = g \text{ xor } \text{true} \text{ xor } \sim g$$

P5 = h

$$d(h) = h \text{ xor } \text{false} \text{ xor } h$$

$$d(h) = h \text{ xor } \text{true} \text{ xor } \sim h$$

3.4. Requisitos de Teste

#	Predicado	Requisito	Pré-requisito
1	p1	$a \vee \sim b$	
2		$\sim a \vee b$	
3	p2	$\sim c \wedge d$	p1 = FALSE
4		$c \wedge \sim d$	
5	p3	$\sim e \wedge f$	p1 = FALSE p2 = FALSE
6		$f \wedge \sim e$	
7	p4	g	
8		$\sim g$	
9	p5	h	
10		$\sim h$	

3.5. Testes Viáveis e Inviáveis

Os testes 5 e 6 são inviáveis, porque o *i* nunca será maior que 10 e o *vat* nunca será igual a 0, respectivamente.

Testes Cobertos	vat	Método	Output Esperado
1	999	isValidVat1()	FALSE
2	1 230 806 345	isValidVat2()	FALSE
3,8,9	152 404 465	isValidVat3()	TRUE
4,7,10	vat = 233693221	isValidVat4()	FALSE

3.6. CACC e RACC

Visto que o GACC só cobre testes baseados em cláusulas, os testes baseado em predicados não existem. Isto é, com CC implica que se uma cláusula de um predicado é TRUE então a outra cláusula do mesmo predicado, não pode ter o mesmo valor:

$p = a \vee b, c = \{a, b\}$

Teríamos uma tabela lógica:

p1 = a b			
Test	a	b	a b
1	T	T	T
2	T	F	T
3	F	T	T
4	F	F	F

Em que os testes cobertos por cláusulas (CC) são T2 e T3, porque quando **a** é TRUE, o **b** é FALSE e vice-versa. Os testes cobertos por predicados (PC), cobrem as situações em que o predicado ou é TRUE ou é FALSE, por exemplo T3(TRUE) e T4(FALSE). No caso da tabela acima, ao recolher o teste em que o **p** é FALSE, entra em contradição com a cobertura por CC porque tanto o **a** como o **b** não tem valores alternados.

Sabendo que PC não é coberto, os testes cobertos por CACC precisam dos testes GACC (que implicam CC) e PC. Os testes aqui realizados nunca irão cobrir na totalidade o método CACC, que por sua vez também não cobrem o RACC. Por exemplo, seguindo o predicado acima:

RACC e CACC

Major clause: a

Minor clause: b

Se a = true então b = false

OUTPUT = true

Se a = false então b = false

OUTPUT = false

Assim podemos provar a premissa: $b(a = \text{true}) = b(a = \text{false})$ e $p1(a = \text{true}) \neq p1(a = \text{false})$

Os caminhos que satisfazem estas condições são **T2** e **T4**

Major clause: b

Minor clause: a

Se b = true então a = false

OUTPUT = true

Se b = false então a = false

OUTPUT = false

Assim podemos provar a premissa: $a(b = \text{true}) = a(b = \text{false})$ e $p1(b = \text{true}) \neq p1(b = \text{false})$

Os caminhos que satisfazem estas condições são **T3** e **T4**

3.7. Testes JUnit

Os testes realizados são apresentados abaixo que correspondem aos testes viáveis apresentados na tabela na [secção 3.1.5](#).

```
public void isValidVatTest1() {  
    Customer customer = mock(Customer.class);  
    when(customer.getVATNumber()).thenReturn(999);  
    assertTrue(CustomerTest.isValidVAT(customer.getVATNumber()));  
}
```

```
public void isValidVatTest2() {  
    Customer customer = mock(Customer.class);  
    when(customer.getVATNumber()).thenReturn(1230806345);  
    assertTrue(CustomerTest.isValidVAT(customer.getVATNumber()));  
}
```

```
public void isValidVatTest3() {  
    Customer customer = mock(Customer.class);  
    when(customer.getVATNumber()).thenReturn(152404465);  
    assertTrue(CustomerTest.isValidVAT(customer.getVATNumber()));  
}
```

```
public void isValidVatTest4() {  
    Customer customer = mock(Customer.class);  
    when(customer.getVATNumber()).thenReturn(233693221);  
    assertTrue(CustomerTest.isValidVAT(customer.getVATNumber()));  
}
```

4. Teste de Sistema à Aplicação Web

Input

- designation
- vatNumber
- phoneNumber
- discountType

Características

C1- designation não está vazio

Bloco 1 - True (base)

Bloco 2 - False

C2- vatNumber é único

Bloco 1 - True (base)

Bloco 2 = False

C3- phoneNumber é numérico

Bloco 1 - True (base)

Bloco 2 - False

C4- Tipo de Desconto

Bloco 1 - NoDiscount (base)

Bloco 2 - ThresholdPercentageDiscount

Bloco 3 - EligibleProductsDiscount

Base Choice Coverage

C1B1-C2B1-C3B1-C4B1 -> todas menos 1

C1B1-C2B1-C3B1-C4B2 -> não satisfaz todas menos 1

C1B1-C2B1-C3B1-C4B3 -> não satisfaz todas menos 1

C1B1-C2B1-C3B2-C4B1 -> não satisfaz todas menos 1

C1B1-C2B1-C3B2-C4B2 -> todas menos 1

C1B1-C2B1-C3B2-C4B3 -> todas menos 1

C1B1-C2B2-C3B1-C4B1 -> não satisfaz todas menos 1

C1B1-C2B2-C3B1-C4B2 -> não satisfaz todas menos 1

C1B1-C2B2-C3B1-C4B3 -> não satisfaz todas menos 1

C1B1-C2B2-C3B2-C4B1 -> todas menos 1

C1B1-C2B2-C3B2-C4B2 -> não satisfaz todas menos 1

C1B1-C2B2-C3B2-C4B3 -> não satisfaz todas menos 1

C1B2-C2B1-C3B1-C4B1 -> não satisfaz todas menos 1

C1B2-C2B1-C3B1-C4B2 -> não satisfaz todas menos 1

C1B2-C2B1-C3B1-C4B3 -> não satisfaz todas menos 1

C1B2-C2B1-C3B2-C4B1 -> todas menos 1

C1B2-C2B1-C3B2-C4B2 -> não satisfaz todas menos 1

C1B2-C2B1-C3B2-C4B3 -> não satisfaz todas menos 1

C1B2-C2B2-C3B1-C4B1 -> não satisfaz todas menos 1

C1B2-C2B2-C3B1-C4B2 -> não tem base

C1B2-C2B2-C3B1-C4B3 -> não tem base

C1B2-C2B2-C3B2-C4B1 -> não satisfaz todas menos 1

C1B2-C2B2-C3B2-C4B2 -> não satisfaz todas menos 1

C1B2-C2B2-C3B2-C4B3 -> não satisfaz todas menos 1

Testes Junit

Com base no critério BCC, realizámos 5 testes ao todo. Antes de cada teste é efetuado um *setup* à base de dados, que para além da ligação, implica um *DELETE_ALL* à tabela *Customers* e adiciona 2 clientes. Para cada método, também efetuamos um *DELETE_ALL*.

Como existe uma série de mensagens, decidimos criar um método auxiliar que percorre a lista de mensagens do *output* e que modifica apenas a variável booleana correspondente a cada uma delas. Assim facilita a interpretação do *output* e a realização dos testes. Contudo, como não tínhamos nenhuma característica que precisasse de resposta da base de dados, decidimos criar um teste que o fizesse. Este teste verifica se o cliente, que está a ser inserido, se existe ou não na base de dados de acordo com o seu *vatNumber*, que tem de ser único.

```
@Test
public final void testIndexPage() {
    beginAt("/");
    assertTitleEquals("SaleSys: Welcome page");
    assertTextPresent("Welcome to SaleSys(c)");
    assertLinkPresentWithExactText("Create client");
    assertLinkPresentWithExactText("Make a sale");
}

public void test1() {
    // Cleans some tables
    db = new DbSetup(dmd, DELETE_ALL);
    db.launch();
    beginAt("/action/clientes/novoCliente");
    assertFormPresent();
    assertButtonPresentWithText("Create client");
    assertFormElementPresent("designacao");
    assertFormElementPresent("npc");
    assertFormElementPresent("telefone");
    assertFormElementPresent("desconto");
    setTextField("designacao", "Tiago Moucho");
    setTextField("npc", "252404467");
    setTextField("telefone", "910889744");
    selectOption("desconto", "No discount");
    submit();
    List<IElement> list = getElementsByXPath("//li");
    checkMessages(list);
    assertTrue(message1);
}

public void test2() {
    // Cleans some tables
    db = new DbSetup(dmd, DELETE_ALL);
    db.launch();
    beginAt("/action/clientes/novoCliente");
    assertFormPresent();
    assertButtonPresentWithText("Create client");
    assertFormElementPresent("designacao");
    assertFormElementPresent("npc");
    assertFormElementPresent("telefone");
    assertFormElementPresent("desconto");
    setTextField("designacao", "Tiago Moucho");
    setTextField("npc", "252404467");
    setTextField("telefone", "91se547d8");
    selectOption("desconto", "Percentage (above threshold)");
    submit();
    List<IElement> list = getElementsByXPath("//li");
    checkMessages(list);
    assertTrue(message6);
    assertTrue(message2);
}
```



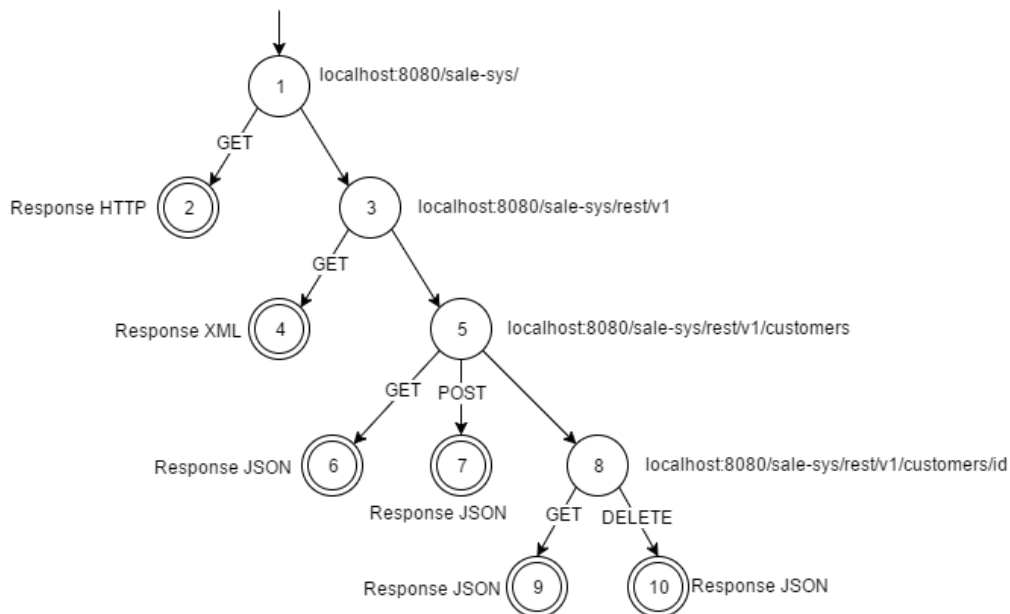
```
public void test3() {  
    // Cleans some tables  
    db = new DbSetup(dmd, DELETE_ALL);  
    db.launch();  
    beginAt("/action/clientes/novoCliente");  
    assertFormPresent();  
    assertButtonPresentWithText("Create client");  
    assertFormElementPresent("designacao");  
    assertFormElementPresent("npc");  
    assertFormElementPresent("telefone");  
    assertFormElementPresent("desconto");  
    setTextField("designacao", "Tiago Moucho");  
    setTextField("npc", "252404467");  
    setTextField("telefone", "91se547d8");  
    selectOption("desconto", "Percentage of total eligible");  
    submit();  
    List<IElement> list = getElementsByXPath("//li");  
    checkMessages(list);  
    assertTrue(message6);  
    assertTrue(message2);  
}
```

```
public void test5() {  
    // Cleans some tables  
    db = new DbSetup(dmd, DELETE_ALL);  
    db.launch();  
    beginAt("/action/clientes/novoCliente");  
    assertFormPresent();  
    assertButtonPresentWithText("Create client");  
    assertFormElementPresent("designacao");  
    assertFormElementPresent("npc");  
    assertFormElementPresent("telefone");  
    assertFormElementPresent("desconto");  
    setTextField("designacao", "");  
    setTextField("npc", "252404467");  
    setTextField("telefone", "91se547d8");  
    selectOption("desconto", "No discount");  
    submit();  
    List<IElement> list = getElementsByXPath("//li");  
    checkMessages(list);  
    assertTrue(message3);  
    assertTrue(message6);  
    assertTrue(message2);  
}
```

```
public void test4() {  
    // Cleans some tables  
    db = new DbSetup(dmd, DELETE_ALL);  
    db.launch();  
    beginAt("/action/clientes/novoCliente");  
    assertFormPresent();  
    assertButtonPresentWithText("Create client");  
    assertFormElementPresent("designacao");  
    assertFormElementPresent("npc");  
    assertFormElementPresent("telefone");  
    assertFormElementPresent("desconto");  
    setTextField("designacao", "Tiago Moucho");  
    setTextField("npc", "252404467");  
    setTextField("telefone", "91se547d8");  
    selectOption("desconto", "No discount");  
    submit();  
    List<IElement> list = getElementsByXPath("//li");  
    checkMessages(list);  
    assertTrue(message2);  
    assertTrue(message6);  
}
```

```
public void bonusTestBD(){  
    beginAt("/action/clientes/novoCliente");  
    assertFormPresent();  
    assertButtonPresentWithText("Create client");  
    assertFormElementPresent("designacao");  
    assertFormElementPresent("npc");  
    assertFormElementPresent("telefone");  
    assertFormElementPresent("desconto");  
    setTextField("designacao", "Tiago Moucho");  
    setTextField("npc", "252404467");  
    setTextField("telefone", "910889744");  
    selectOption("desconto", "No discount");  
    submit();  
    List<IElement> list = getElementsByXPath("//li");  
    checkMessages(list);  
    assertTrue(message9);  
}
```

5. Teste de Sistema à API REST



Pretendemos testar se o resultado de cada caminho Rest corresponde ao suposto *output*, tanto o *statuscode* como o corpo da resposta. Assim, para testar a API Rest decidimos realizar uma cobertura baseada em grafos, nomeadamente *Complete Path Coverage* (CPC). Tendo em conta que a API Rest é construída com base em caminhos, considerámos este mais adequado porque o grafo é a representação abstracta dos caminhos possíveis da API Rest.

Inicialmente pretendíamos usar uma cobertura de Input Space Partitioning, cobertura por escolha de base (BCC) ou por cada bloco de características (ECC). Na maior parte das operações faria sentido, pois recebem *inputs* podendo sendo criadas características adequadas a esses mesmos. Contudo, existe uma operação que não requer um *input*, nem parâmetro de entrada nem variável global, o que faria com que não fosse testada por esses tipos de testes. Adicionalmente, considerámos os testes baseados em Lógica, mas não existe nenhum conteúdo lógico no que toca aos pedidos da API Rest (a nível de *frontend*); ao nível dos testes de cobertura por Syntax há um critério que poderia ser bem adequado ao contexto inserido, Mutating Input Grammars, porém o uso de mutantes exigia uma alteração em pedaços de código que não seriam supostos testar especificamente, visto que o que queremos testar são os caminhos em si e não os métodos.

Para a configuração do DBSetup, considerámos a existência de dois clientes. Esta configuração por vezes é apagada para satisfazer certos requerimentos de um teste. É criada também uma lista dos clientes para corresponder ao *output* recebido da API Rest. Toda esta configuração é feita antes de cada teste, usufruindo do *@Before*, providenciado pelo JUnit.

Dos testes implementados, apenas um não corresponde ao *output* esperado. De acordo com a documentação fornecida, é esperado o valor de 404 para quando o *input* de dados é inválido. No entanto, o valor recebido é 400, falhando então o teste em causa.



Existe uma outra situação em que os testes falham, nomeadamente na operação *delete*. Nesta operação realizámos dois testes, um para quando o *ID* existe e outro para quando o *ID* não existe. No entanto, deparámo-nos com uma falha relativamente ao *role* usado. Por senso comum o *role admin* deveria conseguir ter acesso à operação, aliás até poderia ser exclusivo a este, no entanto as operações só se realizam com o *role basic*. Independentemente desta situação, mantivemos os testes que nos pareciam ser mais adequados realizar (de acordo com o *ID*).