João Marques
8170200

Pedro Pinto
8170262

Ricardo Ferreira
8170279

ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO

Software Unit Testing

Docentes: Doutor Cristóvão Sousa

Dr. Fábio Silva

Alunos: João Marques - 8170200

Ricardo Ferreira - 8170279

Pedro Pinto – 8170262

Índice

Introdução

No âmbito desta cadeira de Engenharia de Software II, pretendemos neste trabalho abordar uma análise de software através de testes de caixa preta utilizando os métodos "**Equivalence Class Partitioning**" e "**Boundary Value Analysis**".

Para cada método estudado temos como objetivo averiguar que para cada estado de entrada válida ocorre um estado de saída apropriado e inversamente para cada estado de entrada inválido, ocorre um estado de saída apropriado.

Objetivo

Hoje em dia é muito útil ter uma boa ferramenta de **teste de software** para quem quer fazer um software. Isto pode custar uma fortuna, e por sua vez não significa que a compra deste tipo de serviço garanta que o software desenvolvido será 100 % livre de **erros**, **bugs** ou até mesmo de **falhas gerais**.

Assim o teste de software é um processo, ou uma série de processos, destinado a garantir que o código respeita as especificações e que não apresenta comportamentos não intencionais, mas sim previsíveis e consistentes, oferecendo o melhor desempenho sem surpresas para os utilizadores. Para efetuar testes podemos ter duas opções: Analisar o código, referidos como testes de **caixa branca**, ou ignorar o código e tratar o sistema como uma **caixa preta**. Neste conceito, que é escolhido para o ambiente de testes proposto no enunciado, são aplicados ao sistema a testar um conjunto de **inputs** e verificado se para cada **output** é igual ao **expected output** de acordo com as especificações.

Referências

- IEEE Standard for Software Unit Testing
- "https://www.moodle.estg.ipp.pt/"
- "https://www.guru99.com/"

Unit Testing Activities

Equivalence Class Partitioning-ECP

Abordamos aqui o método “Equivalence Class Partitioning” que consiste numa técnica destinada a reduzir o número de testes necessários.

Com esta técnica divide-se o domínio de entrada (ou saída) em classes de dados em que o casos de teste podem ser derivados, identificando as classes de equivalência dos argumentos e os estados dos objetos.

Devemos considerar classes de valores válidos e valores inválidos.

getBicycle

Method	Type	Valid Class	Invalid Class	Specific Values
getBicycle	-IDUser: Int -IDDeposit: Int -Credit: Int	Se ID do utilizador se encontrar registado && Se o ID do depósito existir && Se o crédito for suficiente	Se ID do utilizador não se encontrar registado || Se o ID do depósito não existir || Se o crédito não for suficiente	ID utilizador registado>0 ID depósito >0 Crédito>=1
	-Bike: Int	Se existirem bicicletas disponíveis	Se não existirem bicicletas disponíveis	Bicicletas disponíveis>=1
	-rental: Int	Se o utilizador não tiver aluguer ativo	Se o utilizador tiver aluguer ativo	Utilizador aluguer ativo =0

returnBicycle

Method	Type	Valid Class	Invalid Class	Specific Values
returnBicycle	-IDUser: Int -IDDeposit: Int -IDBike: Int	Se ID Utilizador está registado && Se o ID depósito está registado && Se a bicicleta está associada a aluguer ativo	Se ID Utilizador não está registado || Se o ID depósito não está registado || Se a bicicleta não está associada a aluguer ativo	ID User>=0 ID depósito>0 Bicicleta associada a um aluguer ativo: true
	- Lugares livres: Int	Se existirem lugares de entrega livres	Se não existirem lugares de entrega livres	Lugares disponíveis>0

bicycleRentalFee

Method	Type	Valid Class	Invalid Class	Specific Values
bicycleRentalFee	-rentalProgram: Int	Se o valor do rentalProgram for igual a 1	Se o valor do rentalProgram for diferente de 1	rentalProgram>=0
	-nRentals: Int	Se o rentalProgram for igual a 2 && nRentals%10!=0	Se o rentalProgram for diferente de 2 || nRentals%10=0	rentalProgram>=0 nRentals>=0
	-endtime: Int -startime: Int	Se (endtime-startime) for menor ou igual a 10	Se (endtime-startime) for maior do que 10	endtime>=0 startime<=endtime

verifyCredit

Method	Type	Valid Class	Invalid Class	Specific Values
verifyCredit	-IDUser: Int -Crédito: Int	Se o ID do utilizador se encontrar registado && Se o crédito for suficiente	Se o ID do utilizador não se encontrar registado && Se o crédito não for suficiente	ID User>=0 Crédito>=1

addCredit

Method	Type	Valid Class	Invalid Class	Specific Values
addCredit	-IDUser: Int -Amount: ID	Se o ID do utilizador se encontrar registado e amount a adicionar ser maior que zero	Se o ID do utilizador não se encontrar registado e amount a adicionar não ser maior que zero	ID User>=0 Amount>0

registerUser

Method	Type	Valid Class	Invalid Class	Specific Values
registerUser	-IDUser: Int	Se o ID do utilizador não se encontrar registado	Se o ID do utilizador se encontrar registado	ID User>=0 Name!=null rentalProgram 1 ou 2

Boundary Value Analysis -BVA

Abordamos aqui o método “Boundary Value Analysis” que consiste numa técnica focada nos limites do dominio de entrada(ou saída) e imediatamente acima e abaixo.

Nesta técnica baseada na procura de problemas com indices de arrays, decisões, overflow, entre outros.

getBicycle

Test Case	Required System State	Input	Expected Output	Actual Output
1 getBicycle	ID utilizador registado-0 ID depósito 1 Bicicletas disponíveis=1 Utilizador 1= 0 alugueres ativos	ID utilizador- 0 ID depósito-1 Crédito-1 startime=0	Bicicleta associada ao utilizador, é libertado o lugar e retornado o identificador da bicicleta	**Bicicleta associada ao utilizador, é libertado o lugar e retornado o identificador da bicicleta**
2 getBicycle	ID utilizador registado 0 ID depósito 1 Bicicletas disponíveis=1 Utilizador 1= 0 alugueres ativos	**ID utilizador- 1** ID depósito-1 Crédito-1 startime=0	Exception: user does not exist	**Exception: user does not exist**
3 getBicycle	ID utilizador registado 0 ID depósito 1 Bicicletas disponíveis=1 Utilizador 1= 0 alugueres ativos	**ID utilizador= -1** ID depósito=1 Crédito=1 Startime=0	Exception: user does not exist	**Exception: user does not exist**
4 getBicycle	ID utilizador registado 0 ID depósito 1 Bicicletas disponíveis=1 Utilizador 1= 0 alugueres ativos	ID utilizador= 0 **ID depósito=0** Crédito=1 startime=0	Retorna -1	**Retorna -1**
5 getBicycle	ID utilizador registado 0 ID depósito 1 Bicicletas disponíveis=1	ID utilizador= 0 **ID depósito=2** Crédito=1 startime=0	Retorna -1	**Retorna -1**

	Utilizador 1= 0 alugueres ativos			
6 getBicycle	ID utilizador registado 0 ID depósito 1 Bicicletas disponíveis=1 Utilizador 1= 0 alugueres ativos	ID utilizador= 0 ID depósito=1 **Crédito=0** startime=0	Retorna -1	**Retorna -1**
7 getBicycle	ID utilizador registado 0 ID depósito 1 Bicicletas disponíveis=1 Utilizador 1= 0 alugueres ativos	ID utilizador= 0 ID depósito=1 **Crédito=2** startime=0	Bicicleta associada ao utilizador, é libertado o lugar e retornado o identificador da bicicleta	**Bicicleta associada ao utilizador, é libertado o lugar e retornado o identificador da bicicleta**
8 getBicycle	ID utilizador registado 0 ID depósito 1 Bicicletas disponíveis=1 Utilizador 1= 0 alugueres ativos	ID utilizador= 0 ID depósito=1 Crédito=1 **startime= -1**	Retorna -1	**Retorna 1(ID Bike)**
9 getBicycle	ID utilizador registado 0 ID depósito 1 Bicicletas disponíveis=1 Utilizador 1= 0 alugueres ativos	ID utilizador= 0 ID depósito=1 Crédito=1 **startime=1**	Bicicleta associada ao utilizador, é libertado o lugar e retornado o identificador da bicicleta	**Bicicleta associada ao utilizador, é libertado o lugar e retornado o identificador da bicicleta**
10 getBicycle	ID utilizador registado 0 ID depósito 1 **Bicicletas disponíveis=0** Utilizador 1= 0 alugueres ativos	ID utilizador= 0 ID depósito=1 Crédito=1 startime=0	Retorna -1	**Retorna -1**
11 getBicycle	ID utilizador registado 0 ID depósito 1 Bicicletas disponíveis=1 **Utilizador 1= 1 alugueres ativos**	ID utilizador= 0 ID depósito=1 Crédito=1 startime=0	Retorna -1	**Teste Falhado-Retorna 2 (ID Bike)**

Test Case 1 getBicycle

```
@Test
public void test1() throws UserDoesNotExists {
    b.addBicycle( idDeposit: 1,  idLock: 1,  idBike: 1);
    b.addCredit( idUser: 0,  amount: 1);
    b.getBicycle( IDDeposit: 1,  IDUser: 0,  startTime: 0);
    assertTrue(b.getUsers().get(0).getBike().isInUSe());
    assertFalse(b.getBikes().contains(new Bike( IDBike: 1)));
}
```

Neste test case testamos o método getBicycle com argumentos que achamos válidos, esperando que a bicicleta associada ao utilizador, retornado o identificador da bicicleta e libertando o lugar. Tendo o teste passado com sucesso.

Test Case 2 getBicycle

```
@Test
public void test2() {
    b.addBicycle( idDeposit: 1,  idLock: 1,  idBike: 1);
    b.addCredit( idUser: 0,  amount: 1);
    assertThrows(UserDoesNotExists.class,
            () -> {
                b.getBicycle( IDDeposit: 1,  IDUser: 1,  startTime: 0);
            });
}
```

Neste test case testamos o método getBicycle com praticamente os mesmos argumentos, mudando apenas para um utilizador não registado, esperando que dê “Exception: user does not exist”. Tendo o teste passado com sucesso.

Test Case 3 getBicycle

```
@Test
public void test3() {
    b.addBicycle( idDeposit: 1,  idLock: 1,  idBike: 1);
    b.addCredit( idUser: 0,  amount: 1);
    assertThrows(UserDoesNotExists.class,
            () -> {
                b.getBicycle( IDDeposit: 1,  IDUser: -1,  startTime: 0);
            });
}
```

Neste test case testamos o método getBicycle com os mesmos argumentos que achamos válidos do primeiro teste, mudando apenas para um id de utilizador negativo, esperando que dê "Exception: user does not exist". Tendo o teste passado com sucesso.

Test Case 4 getBicycle

```
@Test
public void test4() throws UserDoesNotExists {
    b.addBicycle( idDeposit: 1,  idLock: 1,  idBike: 1);
    b.addCredit( idUser: 0,  amount: 1);
    assertEquals( expected: -1, b.getBicycle( IDDeposit: 0,  IDUser: 0,  startTime: 0));
}
```

Neste test case testamos o método getBicycle com os mesmos argumentos que achamos válidos do primeiro teste, mudando apenas o id do depósito para "0", esperando que retorne "-1". Tendo o teste passado com sucesso.

Test Case 5 getBicycle

```
@Test
public void test5() throws UserDoesNotExists {
    b.addBicycle( idDeposit: 1,  idLock: 1,  idBike: 1);
    b.addCredit( idUser: 0,  amount: 1);
    assertEquals( expected: -1, b.getBicycle( IDDeposit: 2,  IDUser: 0,  startTime: 0));
}
```

Neste test case testamos o método getBicycle com os mesmos argumentos que achamos válidos do primeiro teste, mudando apenas o id do depósito para "2", esperando que retorne "-1". Tendo o teste passado com sucesso.

Test Case 6 getBicycle

```
@Test
public void test6() throws UserDoesNotExists {
    b.addBicycle( idDeposit: 1,  idLock: 1,  idBike: 1);
    assertEquals( expected: -1, b.getBicycle( IDDeposit: 1,  IDUser: 0,  startTime: 0));
}
```

Neste test case testamos o método getBicycle com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas o crédito a “0”, esperando que retorne “-1”. Tendo o teste passado com sucesso.

Test Case 7 getBicycle

```
@Test
public void test7() throws UserDoesNotExists {
    b.addBicycle( idDeposit: 1,  idLock: 1,  idBike: 1);
    b.addCredit( idUser: 0,  amount: 2);
    b.getBicycle( IDDeposit: 1,  IDUser: 0,  startTime: 0);
    assertTrue(b.getUsers().get(0).getBike().isInUSe());
    assertFalse(b.getBikes().contains(new Bike( IDBike: 1)));
}
```

Neste test case testamos o método getBicycle com os mesmo argumentos que achamos válidos no primeiro teste, mudando apenas o crédito para a “2”, esperando que a bicicleta associada ao utilizador, retornado o identificador da bicicleta e libertando o lugar. Tendo o teste passado com sucesso.

Test Case 8 getBicycle

```
@Test
public void test8() throws UserDoesNotExists {
    b.addBicycle( idDeposit: 1,  idLock: 1,  idBike: 1);
    b.addCredit( idUser: 0,  amount: 1);
    assertEquals( expected: -1, b.getBicycle( IDDeposit: 1,  IDUser: 0,  startTime: -1));
}
```

Neste test case testamos o método getBicycle com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas o startime para “-1”, esperando que retorne “-1”. Tendo o teste falhado, retornando “1” que corresponde ao ID da Bike.

Test Case 9 getBicycle

```
@Test
public void test9() throws UserDoesNotExists {
    b.addBicycle( idDeposit: 1,  idLock: 1,  idBike: 1);
    b.addCredit( idUser: 0,  amount: 1);
    b.getBicycle( IDDeposit: 1,  IDUser: 0,  startTime: 1);
    assertTrue(b.getUsers().get(0).getBike().isInUSe());
    assertFalse(b.getBikes().contains(new Bike( IDBike: 1)));

}
```

Neste test case testamos o método getBicycle com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas o startime para “1”, esperando que a bicicleta associada ao utilizador, retornado o identificador da bicicleta e libertando o lugar. Tendo o teste passado com sucesso.

Test Case 10 getBicycle

```
@Test
public void test10() throws UserDoesNotExists {
    b.addCredit( idUser: 0,  amount: 1);
    assertEquals( expected: -1, b.getBicycle( IDDeposit: 1,  IDUser: 0,  startTime: 0));
}
```

Neste test case testamos o método getBicycle com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas as bicicletas disponiveis para “0”, esperando que retorne “-1”. Tendo o teste passado com sucesso.

Test Case 11 getBicycle

```
@Test
public void test11() throws UserDoesNotExists {
    b.addBicycle( idDeposit: 1,  idLock: 1,  idBike: 1);
    b.addCredit( idUser: 0,  amount: 1);
    b.getBicycle( IDDeposit: 1,  IDUser: 0,  startTime: 0);
    b.addBicycle( idDeposit: 2,  idLock: 2,  idBike: 2);
    assertEquals( expected: -1, b.getBicycle( IDDeposit: 2,  IDUser: 0,  startTime: 0));
}
```

Neste test case testamos o método getBicycle com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas os alugueres do utilizador para “1”, esperando que retorne “-1”. Tendo o teste falhado, retornando “2” que corresponde ao ID da Bike.

returnBicycle

Test Case	Required System State	Input	Expected Output	Actual Output
1 returnBicycle	ID User=0 ID depósito=1 Bicicleta associada a um aluguer ativo=true Lugares disponíveis=1 Startime=0	ID User=0 ID depósito=1 endtime=0	Calcula o pagamento através do método bicycleRentalFee	**Calcula o pagamento através do método bicycleRentalFee**
2 returnBicycle	ID User=0 ID depósito=1 Bicicleta associada a um aluguer ativo=true Lugares disponíveis=1 Startime=0	**ID User=1** ID depósito=1 endtime=0	Retorna -1	**Retorna -1**
3 returnBicycle	ID User=0 ID depósito=1 Bicicleta associada a um aluguer ativo=true Lugares disponíveis=1 Startime=0	**ID User=-1** ID depósito=1 endtime=0	Retorna -1	**Retorna -1**
4 returnBicycle	ID User=0 ID depósito=1 Bicicleta associada a um aluguer ativo=true Lugares disponíveis=1 Startime=0	ID User=0 **ID depósito=0** endtime=0	Retorna -1	**Retorna -1**
5 returnBicycle	ID User=0 ID depósito=1 Bicicleta associada a um aluguer ativo=true Lugares disponíveis=1 Startime=0	ID User=0 **ID depósito=2** endtime=0	Retorna -1	**Retorna -1**
6 returnBicycle	ID User=0 ID depósito=1 Bicicleta associada a um aluguer ativo=true Lugares disponíveis=1 Startime=0	ID User=0 ID depósito=1 **endtime=-1**	Retorna -1	**Retorna 1**

7 returnBicycle	ID User=0 ID depósito=1 Bicicleta associada a um aluguer ativo=true Lugares disponíveis=1 Startime=0	ID User=0 ID depósito=1 **endtime= 1**	Calcula o pagamento através do método bicycleRentalFee	**Retorna 1(Valor do cálculo realizado em bicycleRentalFee)**
8 returnBicycle	ID User=0 ID depósito=1 **Bicicleta associada a um aluguer ativo=false** Lugares disponíveis=1 Startime=0	ID User=0 ID depósito=1 endtime=0	Retorna -1	**Retorna -1**
9 returnBicycle	ID User=0 ID depósito=1 Bicicleta associada a um aluguer ativo=true **Lugares disponíveis=0** Startime=0	ID User=0 ID depósito=1 endtime=0	Retorna -1	**Retorna -1**

Test Case 1 returnBicycle

```
@Test
public void testl() throws UserDoesNotExists {
    b.addBicycle( idDeposit: 1,  idLock: 1,  idBike: 1);
    b.addCredit( idUser: 0,  amount: 1);
    b.getBicycle( IDDeposit: 1, IDUser: 0, startTime: 0);
    float help=b.getUsers().get(0).getCredit();
    assertFalse( condition: b.returnBicycle( IDDeposit: 10, IDUser: 0, endTime: 0)==help);
}
```

Neste test case testamos o método returnBicycle com os argumentos que achamos válidos, esperando que calcule o pagamento através do método bicycleRentalFee. Tendo o teste passado com sucesso.

Test Case 2 returnBicycle

```
@Test
public void test2() throws UserDoesNotExists {
    b.addBicycle( idDeposit: 1,  idLock: 1,  idBike: 1);
    b.addCredit( idUser: 0,  amount: 1);
    b.getBicycle( IDDeposit: 1, IDUser: 0, startTime: 0);
    assertEquals( expected: -1, b.returnBicycle( IDDeposit: 1, IDUser: 1, endTime: 0));
}
```

Neste test case testamos o método returnBicycle com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas o ID do utilizador para "1", esperando que retorne "-1". Tendo o teste passado com sucesso.

Test Case 3 returnBicycle

```
@Test
public void test3() throws UserDoesNotExists {
    b.addBicycle( idDeposit: 1,  idLock: 1,  idBike: 1);
    b.addCredit( idUser: 0,  amount: 1);
    b.getBicycle( IDDeposit: 1,  IDUser: 0,  startTime: 0);
    assertEquals( expected: -1, b.returnBicycle( IDDeposit: 1, IDUser: -1, endTime: 0));
}
```

Neste test case testamos o método returnBicycle com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas o ID do utilizador para "-1", esperando que retorne "-1". Tendo o teste passado com sucesso.

Test Case 4 returnBicycle

```
@Test
public void test4() throws UserDoesNotExists {
    b.addBicycle( idDeposit: 1,  idLock: 1,  idBike: 1);
    b.addCredit( idUser: 0,  amount: 1);
    b.getBicycle( IDDeposit: 1,  IDUser: 0,  startTime: 0);
    assertEquals( expected: -1, b.returnBicycle( IDDeposit: 0, IDUser: 0, endTime: 0));
}
```

Neste test case testamos o método returnBicycle com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas o ID do sepósito para “0”, esperando que retorne “-1”. Tendo o teste passado com sucesso.

Test Case 5 returnBicycle

```
@Test
public void test5() throws UserDoesNotExists {
    b.addBicycle( idDeposit: 1,  idLock: 1,  idBike: 1);
    b.addCredit( idUser: 0,  amount: 1);
    b.getBicycle( IDDeposit: 1,  IDUser: 0,  startTime: 0);
    assertEquals( expected: -1, b.returnBicycle( IDDeposit: 2, IDUser: 0, endTime: 0));
}
```

Neste test case testamos o método returnBicycle com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas o ID do depósito para “2”, esperando que retorne “-1”. Tendo o teste passado com sucesso.

Test Case 6 returnBicycle

```
@Test
public void test6() throws UserDoesNotExists {
    b.addBicycle( idDeposit: 1,  idLock: 1,  idBike: 1);
    b.addCredit( idUser: 0,  amount: 1);
    b.getBicycle( IDDeposit: 1,  IDUser: 0,  startTime: 0);
    assertEquals( expected: -1, b.returnBicycle( IDDeposit: 1, IDUser: 0, endTime: -1));
}
```

Neste test case testamos o método returnBicycle com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas o endtime para “-1”, esperando que retorne “-1”. Tendo o teste falhado, retornando “1”.

Test Case 7 returnBicycle

```
@Test
public void test7() throws UserDoesNotExists {
    b.addBicycle( idDeposit: 1,  idLock: 1,  idBike: 1);
    b.addCredit( idUser: 0,  amount: 1);
    b.getBicycle( IDDeposit: 1,  IDUser: 0,  startTime: 0);
    assertEquals( expected: 1, b.returnBicycle( IDDeposit: 1, IDUser: 0, endTime: 1));
}
```

Neste test case testamos o método returnBicycle com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas o endtime para “1”, esperando que calcule o pagamento através do método bicycleRentalFee. Tendo o teste passado com sucesso.

Test Case 8 returnBicycle

```
@Test
public void test8() throws UserDoesNotExists {
    b.addBicycle( idDeposit: 1,  idLock: 1,  idBike: 1);
    b.addCredit( idUser: 0,  amount: 1);
    b.getBicycle( IDDeposit: 1,  IDUser: 0,  startTime: 0);
    b.getBikes().get(0).setInUSe(false);
    assertEquals( expected: -1, b.returnBicycle( IDDeposit: 1, IDUser: 0, endTime: 1));
}
```

Neste test case testamos o método returnBicycle com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas a bicicleta associada a um aluguer ativo para “false”, esperando que retorne “-1”. Tendo o teste passado com sucesso.

Test Case 9 returnBicycle

```
@Test
public void test9() throws UserDoesNotExists {
    b.addBicycle( idDeposit: 1,  idLock: 1,  idBike: 1);
    b.addBicycle( idDeposit: 2, idLock: 2, idBike: 2);
    b.addCredit( idUser: 0,  amount: 1);
    b.getBicycle( IDDeposit: 2, IDUser: 0, startTime: 0);
    assertEquals( expected: -1, b.returnBicycle( IDDeposit: 1, IDUser: 0, endTime: 1));
}
```

Neste test case testamos o método returnBicycle com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas os lugaresdisponiveis para “0”, esperando que retorne “-1”. Tendo o teste passado com sucesso.

bicycleRentalFee

Test Case	Required System State	Input	Expected Output	Actual Output
1 bicycleRentalFee		**rentalProgram=0** endtime=0 startime=0 nRentals=0	Retorna 0	**Retorna 0**
2 bicycleRentalFee		**rentalProgram=1** endtime=0 startime=0 nRentals=0	(endtime-startime)*rentalFee	**(endtime-startime)*rentalFee**
3 bicycleRentalFee		**rentalProgram=2** endtime=0 startime=0 nRentals=0	Retorna 0	**Retorna 0**
4 bicycleRentalFee		**rentalProgram=2** endtime=0 startime=0 **nRentals=1**	rentalFee*(endtime-startime)	**rentalFee*(endtime-startime)**
5 bicycleRentalFee		rentalProgram=2 **endtime=11** startime=0 nRentals=1	10*rentalFee+((endtime-startime)-10)*rentalFee	**10*rentalFee+((endtime-startime)-10)*rentalFee**
6 bicycleRentalFee		rentalProgram=2 endtime=0 startime=0 **nRentals=-1**	Retorna 0	**Retorna 0**
7 bicycleRentalFee		rentalProgram=2 **endtime=0 startime=1** nRentals=0	Retorna 0	**Retorna 0**
8 bicycleRentalFee		**rentalProgram=3** endtime=0 startime=0 nRentals=0	Retorna 0	**Retorna 0**

Test Case 1 bicycleRentalFee

```
@Test
public void test1() {
    assertEquals( expected: 0, b.bicycleRentalFee( rentalProgram: 0,  initTime: 0,  endTime: 1,  nRentals: 0));
}
```

Neste test case testamos o método bicycleRentalFee com os argumentos que achamos válidos, esperando que retorne "0". Tendo o teste passado com sucesso.

Test Case 2 bicycleRentalFee

```
@Test
public void test2() {
    assertEquals( expected: (1 - 0) * b.getRentalFee(), b.bicycleRentalFee( rentalProgram: 1,  initTime: 0,  endTime: 1,  nRentals: 0));
}
```

Neste test case testamos o método bicycleRentalFee com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas o rentalProgram para "1", esperando que retorne o cálculo [(endtime-starttime)*rentalFee]. Tendo o teste passado com sucesso.

Test Case 3 bicycleRentalFee

```
@Test
public void test3() {
    assertEquals( expected: 0, b.bicycleRentalFee( rentalProgram: 2,  initTime: 0,  endTime: 1,  nRentals: 0));
}
```

Neste test case testamos o método bicycleRentalFee com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas o rentalProgram para "2", esperando que retorne "0". Tendo o teste passado com sucesso.

Test Case 4 bicycleRentalFee

```
@Test
public void test4() {
    assertEquals( expected: (1 - 0) * b.getRentalFee(), b.bicycleRentalFee( rentalProgram: 2,  initTime: 0,  endTime: 1,  nRentals: 1));
}
```

Neste test case testamos o método bicycleRentalFee com os mesmos argumentos que achamos válidos no terceiro teste, mudando apenas o nRentals para “1”, esperando que retorne o cálculo [rentalFee*(endtime-starttime)]. Tendo o teste passado com sucesso.

Test Case 5 bicycleRentalFee

```
@Test
public void test5() {
    assertEquals( expected: b.getRentalFee() * 10 + ((11 - 0) - 10) * b.getRentalFee() / 2, b.bicycleRentalFee( rentalProgram: 2,  initTime: 0,  endTime: 11,  nRentals: 1));
}
```

Neste test case testamos o método bicycleRentalFee com os mesmos argumentos que achamos válidos no terceiro teste, mudando apenas o endtime para “11”, esperando que retorne o cálculo [10*rentalFee+((endtime-startime)-10)*rentalFee]. Tendo o teste passado com sucesso.

Test Case 6 bicycleRentalFee

```
@Test
public void test6() {
    assertEquals( expected: 0, b.bicycleRentalFee( rentalProgram: 2,  initTime: 0,  endTime: 1,  nRentals: 0));
}
```

Neste test case testamos o método bicycleRentalFee com os mesmos argumentos que achamos válidos no terceito teste, mudando apenas o nRentals para “-1”, esperando que retorne “0”. Tendo o teste passado com sucesso.

Test Case 7 bicycleRentalFee

```
@Test
public void test7() {
    assertEquals( expected: 0, b.bicycleRentalFee( rentalProgram: 2,  initTime: 1,  endTime: 0,  nRentals: 0));
}
```

Neste test case testamos o método bicycleRentalFee com os mesmos argumentos que achamos válidos no terceiro teste, mudando apenas o starttime para "1", esperando que retorne "0". Tendo o teste passado com sucesso.

Test Case 8 bicycleRentalFee

```
@Test
public void test8() {
    assertEquals( expected: 0, b.bicycleRentalFee( rentalProgram: 3,  initTime: 0,  endTime: 1,  nRentals: 0));
}
```

Neste test case testamos o método bicycleRentalFee com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas o rentalProgram para "3", esperando que retorne "0". Tendo o teste passado com sucesso.

verifyCredit

Test Case	Required System State	Input	Expected Output	Actual Output
1 verifyCredit	ID User=0	ID User=0 Credito=1	Return true	**Return true**
2 verifyCredit	ID User=0	**ID User=1** Crédito=1	Return false	**Return false**
3 verifyCredit	ID User=0	**ID User=-1** Crédito=1	Return false	**Return false**
4 verifyCredit	ID User=0	ID User=0 **Crédito=0**	Return false	**Return false**

Test Case 1 verifyCredit

```
@Test
public void test1() {
    b.addCredit( idUser: 0,  amount: 1);
    assertTrue(b.verifyCredit( IDUser: 0));
}
```

Neste test case testamos o método verifyCredit com os argumentos que achamos válidos, esperando que retorne "true". Tendo o teste passado com sucesso.

Test Case 2 verifyCredit

```
@Test
public void test2() {
    b.addCredit( idUser: 1,  amount: 1);
    assertFalse(b.verifyCredit( IDUser: 0));
}
```

Neste test case testamos o método verifyCredit com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas o ID do utilizador para "1", esperando que retorne "false". Tendo o teste passado com sucesso.

Test Case 3 verifyCredit

```
@Test
public void test3() {
    b.addCredit( idUser: -1,  amount: 1);
    assertFalse(b.verifyCredit( IDUser: -1));
}
```

Neste test case testamos o método verifyCredit com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas o ID do utilizador para "-1", esperando que retorne "false". Tendo o teste passado com sucesso.

Test Case 4 verifyCredit

```
@Test
public void test4() {
    b.addCredit( idUser: 0,  amount: 0);
    assertFalse(b.verifyCredit( IDUser: 0));
}
```

Neste test case testamos o método verifyCredit com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas o crédito para "0", esperando que retorne "false". Tendo o teste passado com sucesso.

addCredit

Test Case	Required System State	Input	Expected Output	Actual Output
1 addCredit	ID User=0	ID User=0 Amount=1	Adiciona amount ao valor de crédito do cliente	**Adiciona amount ao valor de crédito do cliente**
2 addCredit	ID User=0	**ID User=1** Amount=1	Não há alterações no sistema	**Não há alterações no sistema**
3 addCredit	ID User=0	**ID User=-1** Amount=1	Não há alterações no sistema	**Não há alterações no sistema**
4 addCredit	ID User=0	ID User=0 **Amount= -1**	Não há alterações no sistema	**Não há alterações no sistema**

Test Case 1 addCredit

```
@Test
public void test1() {
    b.addCredit( idUser: 0,  amount: 1);
    assertEquals( expected: 1, b.getUsers().get(0).getCredit());
}
```

Neste test case testamos o método addCredit com os argumentos que achamos válidos, esperando que adicione o amount ao valor de crédito do cliente. Tendo o teste passado com sucesso.

Test Case 2 addCredit

```
@Test
public void test2() {
    b.addCredit( idUser: 1,  amount: 1);
    assertFalse(b.getUsers().contains(new User( IDUser: 1,  name: "testUser",  rentalProgram: 1)));
}
```

Neste test case testamos o método addCredit com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas o ID do utilizador para “1”, mas tendo em conta que se trata de um método do tipo void, não estamos á espera de nenhum output, apenas podem existir alterações no Sistema.

Test Case 3 addCredit

```
@Test
public void test3() {
    b.addCredit( idUser: -1,  amount: 1);
    assertFalse(b.getUsers().contains(new User( IDUser: -1,  name: "testUser",  rentalProgram: 1)));
}
```

Neste test case testamos o método addCredit com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas o ID do utilizador para "-1", mas tendo em conta que se trata de um método do tipo void, não estamos á espera de nenhum output, apenas podem existir alterações no Sistema.

Test Case 4 addCredit

```
@Test
public void test4() {
    b.addCredit( idUser: 0,  amount: -1);
    assertFalse( condition: -1 == b.getUsers().get(0).getCredit());
}
```

Neste test case testamos o método addCredit com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas o Amount para "-1", mas tendo em conta que se trata de um método do tipo void, não estamos á espera de nenhum output, apenas podem existir alterações no Sistema.

registerUser

Test Case	Required System State	Input	Expected Output	Actual Output
1 registerUser	ID User=1	ID User=0 Name="Eu" rentalProgram=1	Utilizador é adicionado	**Utilizador é adicionado**
2 registerUser	ID User=1	**ID User=1** Name="Eu" rentalProgram=1	Exception: user already exist	**Exception: user already exist**
3 registerUser	ID User=1	**ID User= -1** Name="Eu" rentalProgram=1	Sem alterações no sistema	**Utilizador é adicionado**

Test Case 1 registerUser

```
@Test
public void test1() throws UserAlreadyExists {
    b.registerUser( IDUser: 1,  name: "testUser0",  rentalProgram: 1);
    assertEquals( expected: 1, b.getUsers().get(1).getIDUser());

}
```

Neste test case testamos o método registerUser com os argumentos que achamos válidos, esperando que o utilizador seja adicionado. Tendo o teste passado com sucesso.

Test Case 2 registerUser

```
@Test
public void test2() {
    assertThrows(UserAlreadyExists.class,
            () -> {
                b.registerUser( IDUser: 0,  name: "testUser",  rentalProgram: 1);
            });
}
```

Neste test case testamos o método registerUser com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas o ID do utilizador para “1”, esperando que dê “Exception: user already exist”. Tendo o teste passado com sucesso.

Test Case 3 registerUser

```
@Test
public void test3() throws UserAlreadyExists {
    b.registerUser( IDUser: -1,  name: "testUser",  rentalProgram: 1);
    assertFalse(  condition: b.getUsers().get(1).getIDUser()==-1);
}
```

Neste test case testamos o método registerUser com os mesmos argumentos que achamos válidos no primeiro teste, mudando apenas o ID do utilizador para “-1”, esperando que não haja . Tendo o teste passado com sucesso.

Conclusão

Este projeto ajudou a desenvolver competências no âmbito dos testes de software e despertou a curiosidade para futuros trabalhos na mesma área.

É sem dúvida importante investir nos testes de software, sendo uma parte importante no desenvolvimento deste mesmo. Para um bom planeamento de testes, o método ECP tem de ser realizado primeiramente para que a aplicação do método BVA seja facilitada.

Para concluir, ao executarmos o nosso planeamento dessa forma foi-nos possível encontrar algumas lacunas no programa que nos foi disponibilizado. Desta forma, fica evidente os benefícios das estratégias de testes devidamente aplicadas.

O código dos testes realizados encontram-se no GitHub em:

"https://github.com/jMarquesss/ES2"