PROJETO ESINF

Gabriel Gonçalves - 1191296

Tiago Leite - 1191369

João Durães - 1211314

Francisco Bogalho - 1211304

António Bernardo - 1210805

INDICE

Introdução

Este projeto tem como objetivo desenvolver um *software* com um conjunto de funcionalidades que permita gerir a informação relativa aos valores de produção de vários tipos de fruto, em vários países, ao longo dos anos.

As funcionalidades requisitadas para a realização deste projeto são:

- ✓ Carregar informação a partir de um dado ficheiro com um certo nome e que contenha a extensão *CSV*;
- ✓ Dado um fruto e uma quantidade de produção, devolver numa lista os países com pelo menos um ano de produção do fruto indicado e com quantidade maior ou igual à indicada. A lista a devolver deve vir ordenada por dois critérios:
  - De forma crescente pelo ano de produção mais baixo em que a produção do fruto indicado foi quantidade maior ou igual à quantidade indicada;
  - Em igualdade do critério anterior, por ordem decrescente da quantidade que foi produzida.
- ✓ Dada uma quantidade de produção, descobrir qual o número mínimo de países que, em conjunto, consegue ter uma quantidade de produção superior à indicada;
- ✓ Dado um fruto, devolver, os países agrupados pelo número máximo de anos consecutivos em que houve crescimento de quantidade de produção do fruto indicado;
- ✓ Dado um determinado país, devolver, numa estrutura adequada, o par de anos, o fruto e o maior valor absoluto da diferença de produção.

Estas funcionalidades devem ser implementadas da forma mais eficiente possível através do desenvolvimento das classes necessárias e com a utilização da *Java Collection Framework*.

Diagrama de classes

Neste diagrama de classes, temos representado um map tree, no mesmo encontramos a relação, entre a *area* e o *item*, de 1 para *, respetivamente, isto significa que dentro de um set a *area* pode ter vários *item's*, mas o *item* só pode pertencer a uma *area*.

Entre o item e o *year* temos, também, uma relação de 1 para *, respetivamente, significando que um item pode ser produzido em vários anos, mas dentro do *set* um ano só pode estar atribuído a um item.

Entre o *year* e o *value* é estabelecida uma relação de 1 para 1, respetivamente, ou seja, um ano só pode ter um valor de produção e esse valor só pode pertencer a esse ano, podendo haver valores de produção repetidos para outros anos diferentes.

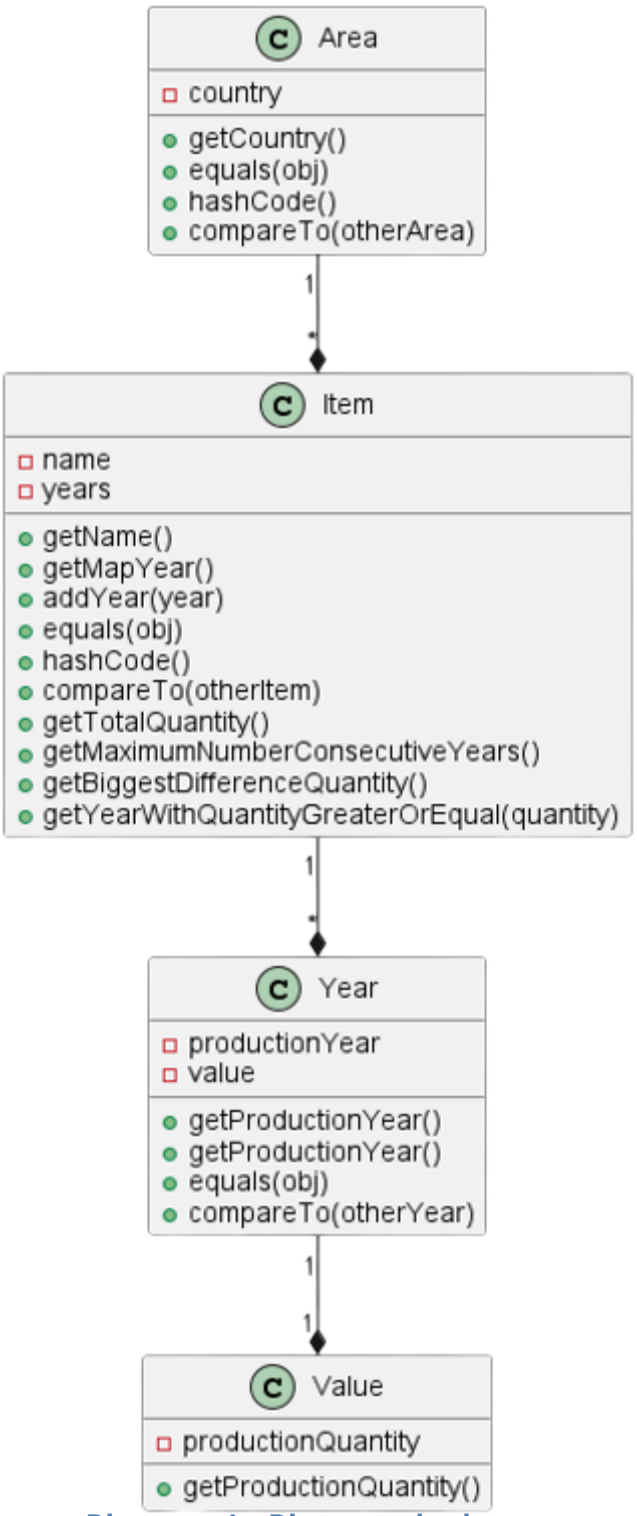

Diagrama 1 - Diagrama de classes

Algoritmos

Para um armazenamento de informação contida no ficheiro *CSV* de forma eficiente para a pesquisa de informação, optamos por criar um *Map* que contêm como *key* a classe *Area* e como *value* um *Set* da classe *Item.* Por sua vez, a classe *Item* contêm um *Set* da classe *Year* que possui a classe *Value* como atributo.

Para uma melhor perceção, a imagem seguinte ilustra como a informação fica armazenada na estrutura adotada.

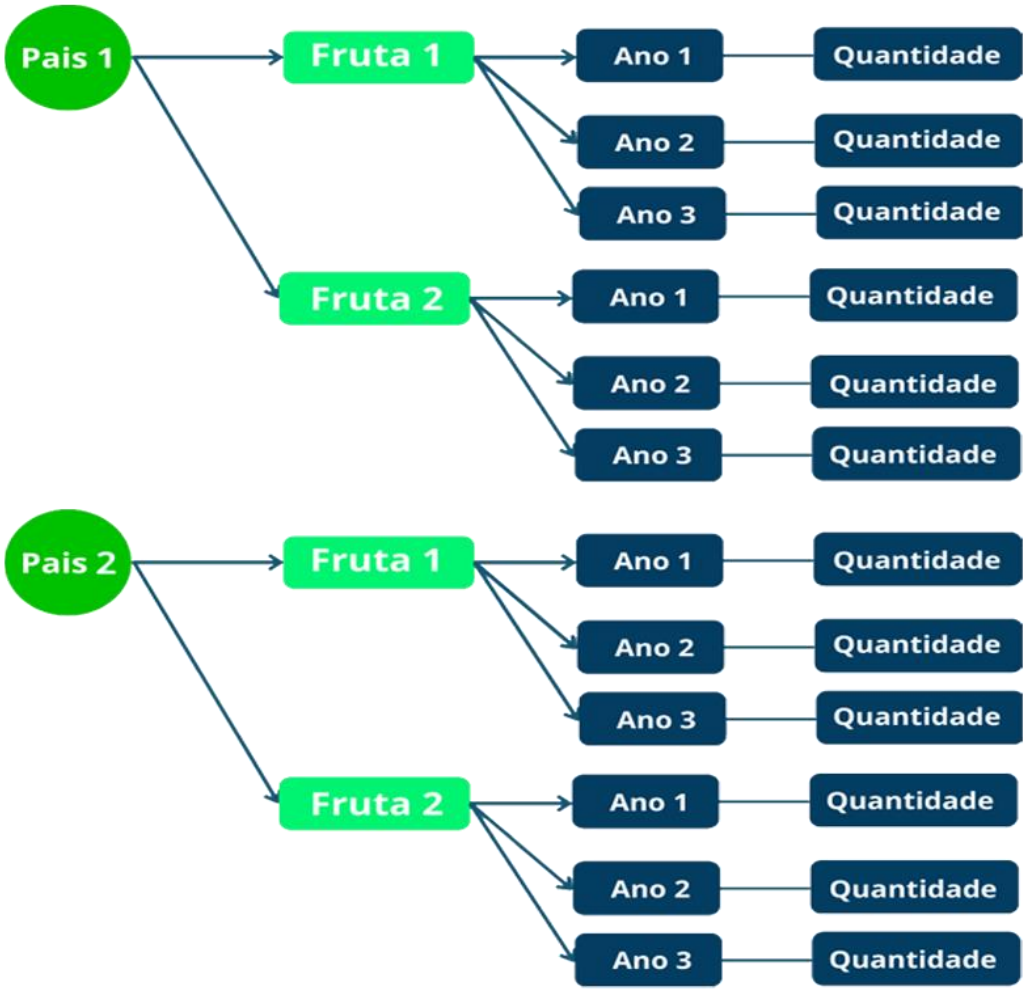

Figura 1 – Simulação da estrutura adotada para armazenar a informação

Para uma melhor estruturação do código, decidiu-se criar uma interface com alguns métodos importantes relativamente ao *Item*.

```
public interface ItemInformation {
    int getTotalQuantity();

    int getMaximumNumberConsecutiveYears();

    Pair<String, Integer> getBiggestDifferenceQuantity();

    Year getYearWithQuantityGreaterOrEqual(int quantity);
}
```

Figura 2 - Excerto de código da interface *ItemInformation*

Na classe *Item* implementou-se a interface referida.

No método *getTotalQuantity()* da classe Item são percorridos todos os objetos do tipo *Year* contidos na estrutura de dados *Set* do Item e retornado o total de quantidade de produção existente.

```
@Override
public int getTotalQuantity() {

    int totalQuantity = 0;

    for (Year year : years) {
        totalQuantity += year.getValue().getProductionQuantity();
    }

    return totalQuantity;
}
```

Figura 3 – Excerto de código da implementação da função *getTotalQuantity()*

No método *getMaximumNumberConsecutiveYears()* da classe *Item* são percorridos todos os objetos do tipo *Year* contidos na estrutura de dados *Set* do *Item* e através de sucessivas comparações entre as quantidades de produção do ano atual e do anterior é obtido o número máximo de anos consecutivos em que houve crescimento de quantidade de produção.

```
@Override
public int getMaximumNumberConsecutiveYears() {
    Year currentYear = null;
    int countTemporal = 1, finalCount = 1;

    for (Year year : years) {
        if (currentYear != null) {
            if (year.getValue().getProductionQuantity() > currentYear.getValue().getProductionQuantity()) {
                countTemporal++;
            } else {
                countTemporal = 1;
            }
        }

        if (countTemporal > finalCount) {
            finalCount = countTemporal;
        }

        currentYear = year;
    }

    return finalCount;
}
```

Figura 4 – Excerto de código da implementação da função *getMaximumNumberConsecutiveYears()*

No método *getYearWithQuantityGreaterOrEqual()* da classe Item é criado um interador para percorrer os objetos do tipo *Year* contidos na estrutura de dados *Set* do *Item* e é comparado a quantidade de produção de cada ano com a quantidade passada por parâmetro. Por fim é devolvido um objeto do tipo *Year*.

```
@Override
public Year getYearWithQuantityGreaterOrEqual(int quantity) {
    Iterator<Year> iterator = years.iterator();

    while (iterator.hasNext()) {

        Year year = iterator.next();
        if (year.getValue().getProductionQuantity() >= quantity) {
            return year;
        }
    }

    return null;
}
```

Figura 5 – Excerto de código da implementação da função *getYearWithQuantityGreaterOrEqual()*

No método *getBiggestDifferenceQuantity* da classe *Item* são percorridos todos os objetos do tipo *Year* contidos na estrutura de dados *Set* do Item e para cada objeto do tipo *year* é feita uma comparação na qual se averigua se o ano atual é diferente de nulo se tal se comprovar falso atribui-se á variável *productionQuantity* o valor absoluto da diferença entre o valor de produção do ano anterior e o valor de produção do ano atual (referente á análise)

Seguidamente é verificado se a variável *productionQuantity* é maior que a variável *differenceProduction* se tal se confirmar atribui-se a *differenceProduction* o valor de *productionQuantity* e atualiza-se a *String* que guarda o par de anos consecutivo com a diferença de produção maior. Depois atualiza o valor de *currentYear*. O método returna de uma instância da classe genérica *Pair* contendo a *String pairYears* e o inteiro *differenceProduction*.

```java
@Override
public Pair<String, Integer> getBiggestDifferenceQuantity() {
    Year currentYear = null;
    int differenceProduction = Integer.MIN_VALUE;
    String pairYears = "";

    for (Year year : years) {
        if (currentYear != null) {
            int productionQuantity = Math.abs(currentYear.getValue().getProductionQuantity() - year.getValue().getProductionQuantity());

            if (productionQuantity > differenceProduction) {
                differenceProduction = productionQuantity;
                pairYears = currentYear.getProductionYear() + "/" + year.getProductionYear();
            }
        }

        currentYear = year;
    }

    return new Pair<>(pairYears, differenceProduction);
}
```

Figura 6 – Excerto de código da implementação da função *getBiggestDifferenceQuantity()*

Relativamente à ordenação pedida no exercício 2, optou-se por criar uma classe que implementa a interface *Comparator*. Como a nossa classe *Year* contém o atributo *Value* por composição, tornou-se simples realizar a ordenação pedida.

Inicialmente é realizada a diferença entre o ano de produção 1 e 2 contido na classe Year para este poder ficar ordenado de forma crescente pelo ano de produção mais baixo.

Caso a diferença entre os dois anos de produção seja 0, significa que estes são iguais e assim deve-se passar para o 2º critério de ordenação, que é por ordem decrescente da quantidade. Para este critério é apenas necessário realizar a diferença entre a quantidade de produção 2 e 1 contida no atributo Value da classe Year.

```
public class YearComparator implements Comparator<Year> {
    @Override
    public int compare(Year year1, Year year2) {
        int compareYear = year1.getProductionYear() - year2.getProductionYear();

        if (compareYear != 0) {
            return compareYear;
        } else {
            return year2.getValue().getProductionQuantity() - year1.getValue().getProductionQuantity();
        }
    }
}
```

Figura 7 – Excerto de código da implementação da função compare da classe "YearComparator"

O método equals da classe *Year,* que se assemelha, fortemente, aos métodos do mesmo nome das classes *Area* e *Item,* começa por verificar se o objeto em analise é igual ao passado em parâmetro, se for o método retorna o valor boleano verdadeiro.

Se o caso anteriormente referido não acontecer verificasse se o objeto em análise é nulo ou se o tipo de classe do objeto em analise não é igual ao objeto passado por parâmetro neste caso é devolvido o valor *boleano* falso. Se o caso anterior não se confirmar é verificado se o ano de produção e a quantidade de produção são iguais aos valores do objeto passado por parâmetro.

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }

    if (obj == null || this.getClass() != obj.getClass()) {
        return false;
    }

    Year otherYear = (Year) obj;

    return productionYear == otherYear.productionYear &&
            getValue().getProductionQuantity() == otherYear.getValue().getProductionQuantity();
}
```

Figura 8 – Excerto de código da função equals da classe Year

Testes unitários

Os testes unitários servem para verificar o grau de qualidade do código para um determinado método a partir da criação de classes de testes. Assim sendo, como testes unitários temos testes para a classe *Area*, *Item*, *Pair*, *Production YearComparator* e para *Year*.

Classe de testes *YearTest*

Testes ao método *equals*

Na classe *YearTest* testamos o método ***equals*** e o método ***compareTo*.** No método ***equals*** verificamos quando temos 2 objetos *Year* iguais ou diferentes.

```
@Test
public void equalsTest1() {
    Year year1 = new Year( productionYear: 2013, new Value( productionQuantity: 1000));
    Year year2 = new Year( productionYear: 2013, new Value( productionQuantity: 1000));

    boolean result = year1.equals(year2);

    Assert.assertTrue(result);
}

/**
 * Testa o método equals com dois objetos Year diferentes.
 */
@Test
public void equalsTest2() {
    Year year1 = new Year( productionYear: 2013, new Value( productionQuantity: 1000));
    Year year2 = new Year( productionYear: 2014, new Value( productionQuantity: 1000));

    boolean result = year1.equals(year2);

    Assert.assertFalse(result);
}
```

Figura 9 – Código relativo ao teste do método *equals* da classe *Year*

Testes ao método *compareTo*

Ainda na classe *YearTest* testamos o método ***compareTo*** verificamos quando:

- Dois objetos *Year* com o atributo ano de produção igual.
- Um objeto *Year* com o ano de produção maior que o de outro objeto.
- Um objeto *Year* com o ano de produção menor que o de outro objeto.

```
@Test
public void compareToTest1() {
    //Anos iguais
    Year year1 = new Year( productionYear: 2013, new Value( productionQuantity: 1000));
    Year year2 = new Year( productionYear: 2013, new Value( productionQuantity: 1500));

    int result = year1.compareTo(year2);
    int expResult = 0;

    Assert.assertEquals(expResult, result);
}
@Test
public void compareToTest2() {
    //Ano 1 maior que Ano 2
    Year year1 = new Year( productionYear: 2014, new Value( productionQuantity: 1000));
    Year year2 = new Year( productionYear: 2013, new Value( productionQuantity: 1500));

    int result = year1.compareTo(year2);
    int expResult = 1;

    Assert.assertEquals(expResult, result);
}
```

```
public void compareToTest3() {
    //Ano 2 maior que Ano 1
    Year year1 = new Year( productionYear: 2013, new Value( productionQuantity: 1000));
    Year year2 = new Year( productionYear: 2014, new Value( productionQuantity: 1500));

    int result = year1.compareTo(year2);
    int expResult = -1;

    Assert.assertEquals(expResult, result);
}
```

Figura 10 - Código relativo ao teste do método compareTo da classe Year

Classe de testes *PairTest*

Testes ao método equals

Na classe *PairTest* testamos o método ***equals*** e o método ***compareTo*.** No método ***equals*** verificamos se temos 2 objetos *Pair* iguais ou diferentes.

```java
/**
 * Testa o método equals com dois objetos Pair iguais.
 */
@Test
public void equalsTest1() {
    Pair pair1 = new Pair("Portugal", 10);
    Pair pair2 = new Pair("Portugal", 10);
    boolean result = pair1.equals(pair2);
    Assert.assertTrue(result);
}

/**
 * Testa o método equals com dois objetos Pair diferentes.
 */
@Test
public void equalsTest2() {
    Pair pair1 = new Pair("Portugal", 10);
    Pair pair2 = new Pair("Espanha", 10);
    boolean result = pair1.equals(pair2);
    Assert.assertFalse(result);
}
```

Figura 11 - Código relativo ao teste do método *equals* da classe *Pair*

Testes ao método *compareTo*

Ainda na classe *PairTest* testamos o método ***compareTo*** verificamos quando:

- Dois objetos *Pair* com o atributo *value* igual.
- Um objeto *Pair* com o *value* inferior que o de outro objeto.
- Um objeto *Pair* com o *value* superior que o de outro objeto.

```java
@Test
public void compareToTest1() {
    Pair pair1 = new Pair("Portugal", 10);
    Pair pair2 = new Pair("Portugal", 10);

    int result = pair1.compareTo(pair2);
    int expResult = 0;
    Assert.assertEquals(expResult,result);
}
```

```java
public void compareToTest2() {
    Pair pair1 = new Pair("Portugal", 10);
    Pair pair2 = new Pair("Espanha", 20);

    int result = pair1.compareTo(pair2);
    int expResult = -10;
    Assert.assertEquals(expResult,result);
}
```

```java
public void compareToTest3() {
    Pair pair2 = new Pair("Portugal", 10);
    Pair pair1 = new Pair("Espanha", 20);

    int result = pair1.compareTo(pair2);
    int expResult = 10;
    Assert.assertEquals(expResult,result);
}
```

Figura 12 - Código relativo ao teste do método *compareTo* da classe *Pair*

Classe de testes *YearComparatorTest*

Testes ao método *compare*

Na classe *YearComparatorTest* testamos o método ***compare***. No método ***compare*** verificamos se a função devolve com anos diferentes os valores ordenados pelos critérios pedidos.

```java
@Test
public void compareTest2() {
    //Anos com anos iguais e valores diferentes
    Year year1 = new Year( productionYear: 1990, new Value( productionQuantity: 1000));
    Year year2 = new Year( productionYear: 1990, new Value( productionQuantity: 1500));
    Year year3 = new Year( productionYear: 1990, new Value( productionQuantity: 1800));
    Year year4 = new Year( productionYear: 1993, new Value( productionQuantity: 1500));
    Year year5 = new Year( productionYear: 1993, new Value( productionQuantity: 1000));

    ArrayList<Year> result = new ArrayList<>();
    result.add(year1);
    result.add(year2);
    result.add(year3);
    result.add(year4);
    result.add(year5);

    result.sort(new YearComparator());

    ArrayList<Year> expResult = new ArrayList<>();
    expResult.add(year3);
    expResult.add(year2);
    expResult.add(year1);
    expResult.add(year4);
    expResult.add(year5);

    Assert.assertEquals(expResult, result);
}
```

```java
@Test
public void compareTest1() {
    Year year1 = new Year( productionYear: 2013, new Value( productionQuantity: 1000));
    Year year2 = new Year( productionYear: 1990, new Value( productionQuantity: 1500));
    Year year3 = new Year( productionYear: 2012, new Value( productionQuantity: 1000));
    Year year4 = new Year( productionYear: 1916, new Value( productionQuantity: 1500));
    Year year5 = new Year( productionYear: 2000, new Value( productionQuantity: 1000));

    ArrayList<Year> result = new ArrayList<>();
    result.add(year1);
    result.add(year2);
    result.add(year3);
    result.add(year4);
    result.add(year5);

    result.sort(new YearComparator());

    ArrayList<Year> expResult = new ArrayList<>();
    expResult.add(year4);
    expResult.add(year2);
    expResult.add(year5);
    expResult.add(year3);
    expResult.add(year1);

    Assert.assertEquals(expResult, result);
}
```

Figura 13 - Excerto do código ao teste do método compare da classe *YearComparator*

Classe de testes *AreaTest*

Na classe *AreaTest* testamos os métodos ***compareTo*** e ***equals***.

Testes ao método *equals*

No método ***equals*** verificamos se temos 2 objetos *Area* iguais ou diferentes.

```java
@Test
public void equalsTest1() {
    Area area1 = new Area( country: "Portugal");
    Area area2 = new Area( country: "Portugal");
    boolean result = area1.equals(area2);
    Assert.assertTrue(result);
}
```

```java
@Test
public void equalsTest2() {
    Area area1 = new Area( country: "Portugal");
    Area area2 = new Area( country: "Espanha");
    boolean result = area1.equals(area2);
    Assert.assertFalse(result);
}
```

Figura 14 - Excerto do código da classe AreaTest

Testes ao método compareTo

Ainda na classe *AreaTest* testamos o método **compareTo** verificamos quando:

- Dois objetos *Area* com o atributo *area* igual.
- Um objeto *Area* com o atributo *area* inferior que o de outro objeto.
- Um objeto *Area* com o atributo *area* superior que o de outro objeto.

```java
@Test
public void compareToTest1() {
    Area area1 = new Area( country: "Portugal");
    Area area2 = new Area( country: "Portugal");

    int result = area1.compareTo(area2);
    int expResult = 0;
    Assert.assertEquals(expResult,result);
}
```

```java
@Test
public void compareToTest2() {
    Area area1 = new Area( country: "Albania");
    Area area2 = new Area( country: "Camaroes");

    int result = area1.compareTo(area2);
    int expResult = -2;
    Assert.assertEquals(expResult,result);
}
```

```java
@Test
public void compareToTest3() {
    Area area1 = new Area( country: "Camaroes");
    Area area2 = new Area( country: "Albania");

    int result = area1.compareTo(area2);
    int expResult = 2;
    Assert.assertEquals(expResult,result);
}
```

Figura 15 - Excerto código da Class AreaTest

Classe de testes *ItemTest*

Testes ao método *equals*

Na classe *ItemTest* realizou-se o teste a todos os métodos da classe Item. No método ***equals*** verificamos se temos 2 objetos *Area* iguais ou diferentes.

```
@Test
public void equalsTest1() {
    Item item1 = new Item( name: "Apple");
    Item item2 = new Item( name: "Apple");

    boolean result = item1.equals(item2);

    Assert.assertTrue(result);
}
```

```
@Test
public void equalsTest2() {
    Item item1 = new Item( name: "Apple");
    Item item2 = new Item( name: "Bananas");

    boolean result = item1.equals(item2);

    Assert.assertFalse(result);
}
```

Figura 16 - Excerto do teste do código do método equals da classe *Item*

Testes ao método *compareTo*

Ainda na classe *ItemTest* testamos o método ***compareTo*** verificamos quando:

- Dois objetos Item com o atributo item igual.
- Um objeto Item com o atributo item inferior que o de outro objeto.
- Um objeto Item com o atributo item superior que o de outro objeto.

```
@Test
public void compareToTest1() {
    Item item1 = new Item( name: "Apple");
    Item item2 = new Item( name: "Apple");

    int result = item1.compareTo(item2);
    int expResult = 0;

    Assert.assertEquals(expResult, result);
}
```

```
@Test
public void compareToTest2() {
    Item item1 = new Item( name: "Bananas");
    Item item2 = new Item( name: "Apples");

    int result = item1.compareTo(item2);
    int expResult = 1;

    Assert.assertEquals(expResult, result);
}
```

```
@Test
public void compareToTest3() {
    Item item1 = new Item( name: "Apple");
    Item item2 = new Item( name: "Bananas");

    int result = item1.compareTo(item2);
    int expResult = -1;

    Assert.assertEquals(expResult, result);
```

Figura 17 - Excerto do teste do código do método compare da classe *Item*

Testes ao método getTotalQuantity

Ainda na classe *ItemTest* testamos o método **getTotalQuantity** verificamos se o método devolve os valores que realmente correspondem à realidade.

```
@Test
public void getTotalQuantityTest1() {
    Item item1 = new Item( name: "Apple");
    item1.addYear(new Year( productionYear: 2015, new Value( productionQuantity: 19345)));
    item1.addYear(new Year( productionYear: 2016, new Value( productionQuantity: 1300)));
    item1.addYear(new Year( productionYear: 2017, new Value( productionQuantity: 1905)));
    item1.addYear(new Year( productionYear: 2018, new Value( productionQuantity: 2345)));
    item1.addYear(new Year( productionYear: 2019, new Value( productionQuantity: 4345)));

    int result = item1.getTotalQuantity();
    int expResult = 29240;

    Assert.assertEquals(expResult, result);
}
```

```
@Test
public void getTotalQuantityTest2() {
    Item item1 = new Item( name: "Apple");
    item1.addYear(new Year( productionYear: 2018, new Value( productionQuantity: 0)));
    item1.addYear(new Year( productionYear: 2019, new Value( productionQuantity: 0)));

    int result = item1.getTotalQuantity();
    int expResult = 0;

    Assert.assertEquals(expResult, result);
}
```

Figura 18 - Excerto do código da classe *ItemTest*

Testes ao método getMaximumNumberConsecutiveYears

Ainda na classe *ItemTest* testamos o método **getMaximumNumberConsecutiveYears** verificamos se o método devolve os valores que realmente correspondem à realidade.

```
@Test
public void getMaximumNumberConsecutiveYearsTest1() {
    Item item1 = new Item( name: "Apple");
    item1.addYear(new Year( productionYear: 2015, new Value( productionQuantity: 145)));
    item1.addYear(new Year( productionYear: 2016, new Value( productionQuantity: 1300)));
    item1.addYear(new Year( productionYear: 2017, new Value( productionQuantity: 905)));
    item1.addYear(new Year( productionYear: 2018, new Value( productionQuantity: 2345)));
    item1.addYear(new Year( productionYear: 2019, new Value( productionQuantity: 3345)));

    int result = item1.getMaximumNumberConsecutiveYears();
    int expResult = 3;

    Assert.assertEquals(expResult, result);
}
```

```
@Test
public void getMaximumNumberConsecutiveYearsTest2() {
    Item item1 = new Item( name: "Apple");
    item1.addYear(new Year( productionYear: 2015, new Value( productionQuantity: 23095)));
    item1.addYear(new Year( productionYear: 2016, new Value( productionQuantity: 1345)));
    item1.addYear(new Year( productionYear: 2017, new Value( productionQuantity: 905)));
    item1.addYear(new Year( productionYear: 2018, new Value( productionQuantity: 803)));
    item1.addYear(new Year( productionYear: 2019, new Value( productionQuantity: 730)));

    int result = item1.getMaximumNumberConsecutiveYears();
    int expResult = 1;

    Assert.assertEquals(expResult, result);
}
```

Figura 19 - Excerto do código da classe *ItemTest*

Testes ao método *getBiggestDifferenceQuantity*

Ainda na classe *ItemTest* testamos o método ***getBiggestDifferenceQuantity*** verificamos se o método devolve os valores que realmente correspondem à realidade.

```
@Test
public void getBiggestDifferenceQuantityTest1() {
    Item item1 = new Item( name: "Apple");
    item1.addYear(new Year( productionYear: 2015, new Value( productionQuantity: 19345)));
    item1.addYear(new Year( productionYear: 2016, new Value( productionQuantity: 1300)));
    item1.addYear(new Year( productionYear: 2017, new Value( productionQuantity: 905)));
    item1.addYear(new Year( productionYear: 2018, new Value( productionQuantity: 2345)));
    item1.addYear(new Year( productionYear: 2019, new Value( productionQuantity: 4345)));

    Pair result = item1.getBiggestDifferenceQuantity();
    Pair expResult = new Pair("2015/2016", 18045);

    Assert.assertEquals(expResult, result);
}
```

```
@Test
public void getBiggestDifferenceQuantityTest2() {
    Item item1 = new Item( name: "Apple");
    item1.addYear(new Year( productionYear: 2015, new Value( productionQuantity: 0)));
    item1.addYear(new Year( productionYear: 2016, new Value( productionQuantity: 0)));
    item1.addYear(new Year( productionYear: 2017, new Value( productionQuantity: 0)));
    item1.addYear(new Year( productionYear: 2018, new Value( productionQuantity: 0)));
    item1.addYear(new Year( productionYear: 2019, new Value( productionQuantity: 0)));

    Pair result = item1.getBiggestDifferenceQuantity();
    Pair expResult = new Pair("2015/2016", 0);

    Assert.assertEquals(expResult, result);
}
```

Figura 20 - Excerto do código da classe *ItemTest*

Testes ao método *getYearWithQuantityGreaterOrEqual*

Ainda na classe *ItemTest* testamos o método ***getYearWithQuantityGreaterOrEqual*** verificamos se o método devolve os valores que realmente correspondem à realidade.

```
@Test
public void getYearWithQuantityGreaterOrEqualTest1() {
    Item item1 = new Item( name: "Apple");
    item1.addYear(new Year( productionYear: 2015, new Value( productionQuantity: 345)));
    item1.addYear(new Year( productionYear: 2016, new Value( productionQuantity: 1300)));
    item1.addYear(new Year( productionYear: 2017, new Value( productionQuantity: 905)));
    item1.addYear(new Year( productionYear: 2018, new Value( productionQuantity: 2345)));
    item1.addYear(new Year( productionYear: 2019, new Value( productionQuantity: 4345)));

    Year result = item1.getYearWithQuantityGreaterOrEqual(1000);
    Year expResult = new Year( productionYear: 2016, new Value( productionQuantity: 1300));

    Assert.assertEquals(expResult, result);
}
```

```
@Test
public void getYearWithQuantityGreaterOrEqualTest2() {
    Item item1 = new Item( name: "Apple");
    item1.addYear(new Year( productionYear: 2015, new Value( productionQuantity: 345)));
    item1.addYear(new Year( productionYear: 2016, new Value( productionQuantity: 1300)));
    item1.addYear(new Year( productionYear: 2017, new Value( productionQuantity: 905)));
    item1.addYear(new Year( productionYear: 2018, new Value( productionQuantity: 2345)));
    item1.addYear(new Year( productionYear: 2019, new Value( productionQuantity: 4345)));

    Year result = item1.getYearWithQuantityGreaterOrEqual(5000);

    Assert.assertNull(result);
}
```

Figura 21 - Excerto do código da classe *ItemTest*

Classe de testes *ProductionTest*

Na classe *ProductionTest* realizou-se o teste a todos os métodos da classe *Production*.

Nesta classe, antes de todos os testes serem realizados, é feita a leitura dos dados presentes no ficheiro de dados grande e também do pequeno, cada um para o seu próprio objeto de *Production*. Com esta leitura, estamos também indiretamente a realizar o teste à classe *readFile*.

```
@BeforeClass
public static void beforeClass() throws Exception {
    production.readFile(FILE_NAME);
    smallProduction.readFile(SMALL_FILE_NAME);
}
```

Figura 22 - Excerto do código do método chamado antes de todos os testes da classe *ProductionTest*

Testes ao método readFile

Foram realizados 4 testes diferentes a este método de forma a garantir o seu bom funcionamento.

O primeiro teste passou por verificar se o método lançava corretamente a exceção por tentativa de passar uma *String* para o tipo inteiro.

```
@Test(expected = InvalidLineException.class)
public void readFileTest1() throws FileNotFoundException {
    Production p = new Production();
    p.readFile( filePath: "files/ficheiro_teste_com_erro.csv");
}
```

Figura 23 - Excerto do código do 1º teste para o método readFile

O segundo teste passou por validar se o método lançava corretamente a exceção quando não encontrava o ficheiro no caminho inserido.

```
@Test(expected = FileNotFoundException.class)
public void readFileTest2() throws FileNotFoundException {
    Production p = new Production();
    p.readFile( filePath: "files/aaaaa.csv");
}
```

***Figura* 24 - Excerto do código do 2º teste para o método readFile**

O terceiro teste passou por tentar passar o caminho como *null* para verificar se o método conseguia lidar com esse valor.

```
@Test(expected = FileNotFoundException.class)
public void readFileTest3() throws FileNotFoundException {
    Production p = new Production();
    p.readFile( filePath: null);
}
```

Figura 25 - Excerto do terceiro teste do código de teste do método readFile

O quarto teste passou por validar se as informações contidas num dado ficheiro eram lidas e colocadas corretamente no mapa de dados.

```
@Test
public void readFileTest4() throws FileNotFoundException {
    Production p = new Production();
    p.readFile( filePath: "files/ficheiro_teste_sem_erro.csv");
    Map<Area, Set<Item>> actual = p.getProductionPerArea();

    Map<Area, Set<Item>> expected = new HashMap<>();
    Set<Item> itemSet = new TreeSet<>();

    Item item = new Item( name: "Apples");
    Value value = new Value( productionQuantity: 15100);
    Year yearOne = new Year( productionYear: 1961, value);
    Year yearTwo = new Year( productionYear: 1962, value);
    item.addYear(yearOne);
    item.addYear(yearTwo);
    itemSet.add(item);
    expected.put(new Area( country: "Afghanistan"), itemSet);

    itemSet = new TreeSet<>();
    item = new Item( name: "Apples");
    Value valueTwo = new Value( productionQuantity: 10004);
    Value valueThree = new Value( productionQuantity: 8039);
    yearOne = new Year( productionYear: 1965, valueTwo);
    yearTwo = new Year( productionYear: 1962, valueThree);
    item.addYear(yearOne);
    item.addYear(yearTwo);
    itemSet.add(item);
    expected.put(new Area( country: "Albania"), itemSet);

    Assert.assertEquals(expected, actual);
}
```

Figura 26 - Excerto do quarto teste do código de teste do método readFile

Testes ao método higherValueDifferenceProduction

Foram realizados três testes diferentes para garantir o correto funcionamento deste método.

O primeiro teste serve para verificar se o método devolve um mapa vazio quando passado um país inexistente no mapa que contém toda a informação importada do ficheiro.

```
@Test
public void higherValueDifferenceProductionTest1() {
    Map<Item, Pair<String, Integer>> result = production.higherValueDifferenceProduction(production.getProductionPerArea(), new Area( country: "San Marino"));

    Assert.assertTrue(result.isEmpty());
}
```

Figura 27 - Excerto do código do 1º teste do método higherValueDifferenceProduction

O segundo teste serve para verificar se o método devolve para o país "Malawi" que contém duas frutas, os valores que realmente correspondem à realidade.

```
@Test
public void higherValueDifferenceProductionTest2() {
    Map<Item, Pair<String, Integer>> result = production.higherValueDifferenceProduction(production.getProductionPerArea(), new Area( country: "Malawi"));

    Map<Item, Pair<String, Integer>> expResult = new TreeMap<>();
    expResult.put(new Item( name: "Apples"), new Pair<>("2013/2014", 3203));
    expResult.put(new Item( name: "Bananas"), new Pair<>("1998/1999", 207000));

    Assert.assertEquals(expResult, result);
}
```

Figura 28 - Excerto do código do 2º teste do método higherValueDifferenceProduction

O terceiro teste serve para verificar se o método devolve para o país "Portugal" que contém quatro frutas, os valores que realmente correspondem à realidade.

```
@Test
public void higherValueDifferenceProductionTest3() {
    Map<Item, Pair<String, Integer>> result = production.higherValueDifferenceProduction(production.getProductionPerArea(), new Area( country: "Portugal"));

    Map<Item, Pair<String, Integer>> expResult = new TreeMap<>();
    expResult.put(new Item( name: "Apples"), new Pair<>("1985/1986", 143460));
    expResult.put(new Item( name: "Bananas"), new Pair<>("1979/1980", 10579));
    expResult.put(new Item( name: "Blueberries"), new Pair<>("2018/2019", 4100));
    expResult.put(new Item( name: "Cherries"), new Pair<>("2019/2020", 12760));

    Assert.assertEquals(expResult, result);
}
```

Figura 29 - Excerto do código do 3º teste do método higherValueDifferenceProduction

Testes ao método countryWithYearAndValue

Para este método, foram desenvolvidos três testes diferentes de forma a verificar que tudo funciona como é esperado.

O primeiro teste verifica se o método devolve com a utilização da fruta "Apples" e com a quantidade 2700000, os valores que realmente correspondem à realidade.

```
@Test
public void countryWithYearAndValueTest1() {
    ArrayList<Area> result = production.countryWithYearAndValue(production.getProductionPerArea(), new Item( name: "Apples"), new Value( productionQuantity: 2700000));

    ArrayList<Area> expResult = new ArrayList<>();
    expResult.add(new Area( country: "France"));
    expResult.add(new Area( country: "United States of America"));
    expResult.add(new Area( country: "USSR"));
    expResult.add(new Area( country: "Germany"));
    expResult.add(new Area( country: "China, mainland"));
    expResult.add(new Area( country: "Iran (Islamic Republic of)"));
    expResult.add(new Area( country: "Poland"));
    expResult.add(new Area( country: "Türkiye"));
    expResult.add(new Area( country: "India"));

    Assert.assertEquals(expResult, result);
}
```

Figura 30 - Excerto do primeiro teste do código de teste do método getTotalValueList

O segundo teste serve para verificar se o método devolve um ArrayList vazio quando passado um fruto inexistente no mapa.

```
@Test
public void countryWithYearAndValueTest2() {
    ArrayList<Area> result = production.countryWithYearAndValue(production.getProductionPerArea(), new Item( name: "Coconuts"), new Value( productionQuantity: 2700000));

    Assert.assertTrue(result.isEmpty());
}
```

Figura 31 - Excerto do segundo teste do código de teste do método getTotalValueList

O terceiro teste foi desenvolvido para verificar se o método devolve um ArrayList vazio quando passado uma quantidade de produção inexistente.

```
@Test
public void countryWithYearAndValueTest3() {
    ArrayList<Area> result = production.countryWithYearAndValue(production.getProductionPerArea(), new Item( name: "Apples"), new Value( productionQuantity: 100000000));

    Assert.assertTrue(result.isEmpty());
}
```

Figura 32 - Excerto do terceiro teste do código de teste do método getTotalValueList

Testes ao método maximumNumberConsecutiveYears

Para este método foram desenvolvidos dois testes diferentes de forma a tornar fiável a utilização deste método.

O primeiro teste serve para verificar se o método devolve para o fruto "Brazil nuts, with shell" os valores que realmente correspondem à realidade.

```
@Test
public void maximumNumberConsecutiveYearsTest1() {
    Map<Integer, Set<Area>> result = production.maximumNumberConsecutiveYears(production.getProductionPerArea(), new Item( name: "Brazil nuts, with shell"));

    Map<Integer, Set<Area>> expResult = new TreeMap<>();
    Set<Area> areaSet1 = new TreeSet<>();
    areaSet1.add(new Area( country: "Côte d'Ivoire"));
    areaSet1.add(new Area( country: "Morocco"));
    expResult.put(2, areaSet1);
    Set<Area> areaSet2 = new TreeSet<>();
    areaSet2.add(new Area( country: "Peru"));
    expResult.put(5, areaSet2);
    Set<Area> areaSet3 = new TreeSet<>();
    areaSet3.add(new Area( country: "Bolivia (Plurinational State of)"));
    areaSet3.add(new Area( country: "Brazil"));
    expResult.put(6, areaSet3);


    Assert.assertEquals(expResult, result);
}
```

Figura 33 - Excerto do código do 1º teste ao método maximumNumberConsecutiveYears

O segundo teste serve para verificar se o método devolve um mapa vazio quando passado um fruto inexistente.

```
@Test
public void maximumNumberConsecutiveYearsTest2() {
    Map<Integer, Set<Area>> result = production.maximumNumberConsecutiveYears(production.getProductionPerArea(), new Item( name: "Pera"));

    Assert.assertTrue(result.isEmpty());
}
```

Figura 34 - Excerto do código do 2º teste ao método maximumNumberConsecutiveYears

Testes ao método getTotalValueList

Para este método foram desenvolvidos três testes diferentes para garantir o seu bom funcionamento.

O primeiro teste serve para testar se o tamanho da lista devolvida pelo método é do tamanho correto quando passado um mapa vazio, e também quando é passado um valor null.

```
@Test
public void getTotalValueListTest1() {

    Map<Area, Set<Item>> pd = new HashMap<>();
    ArrayList<Pair<Area, Integer>> totalValueList = production.getTotalValueList(pd);
    int actual = totalValueList.size();
    int expected = 0;
    Assert.assertEquals(expected, actual);

    totalValueList = production.getTotalValueList( production: null);
    Assert.assertNull(totalValueList);

}
```

Figura 35 - Excerto do código do 1º teste ao método getTotalValueList

O segundo teste serve para perceber se os tamanhos dos ArrayList gerados pelos valores dos mapas, criados através da leitura do ficheiro pequeno e outro através do ficheiro grande, estão corretos.

```
@Test
public void getTotalValueListTest2() {

    ArrayList<Pair<Area, Integer>> totalValueList = smallProduction.getTotalValueList(smallProduction.getProductionPerArea());
    int actual = totalValueList.size();
    int expected = 2;
    Assert.assertEquals(expected, actual);

    totalValueList = production.getTotalValueList(production.getProductionPerArea());
    actual = totalValueList.size();
    expected = 187;
    Assert.assertEquals(expected, actual);

}
```

Figura 36 - Excerto do código do 2º teste ao método getTotalValueList

O terceiro teste serve para verificar se os valores que são colocados no ArrayList através da utilização dos dados presentes no ficheiro pequeno estão corretos.

```
@Test
public void getTotalValueListTest3() {

    ArrayList<Pair<Area, Integer>> expected = new ArrayList<>();
    expected.add(new Pair<>(new Area( country: "Portugal"), 6176185));
    expected.add(new Pair<>(new Area( country: "Spain"), 21978760));
    ArrayList<Pair<Area, Integer>> actual = smallProduction.getTotalValueList(smallProduction.getProductionPerArea());
    Assert.assertEquals(expected, actual);

}
```

Figura 37 - Excerto do terceiro teste do código de teste do método getTotalValueList

Teste ao método getMinNumberOfKeysHigherThanQuantity

Para garantir o bom funcionamento deste método, foram desenvolvidos dois testes diferentes.

O primeiro teste serve para verificar se é devolvida a contagem correta de países mínimos para a menor quantidade possível (0), quando se passa um ArrayList nulo e um vazio.

```
@Test
public void getMinNumberOfKeysHigherThanQuantityTest1() {

    long quantity = -1;
    int actual = production.getMinNumberOfKeysHigherThanQuantity( list: null, quantity);
    int expected = 0;
    Assert.assertEquals(expected, actual);

    ArrayList<Pair<Area, Integer>> list = new ArrayList<>();
    actual = production.getMinNumberOfKeysHigherThanQuantity(list, quantity);
    Assert.assertEquals(expected, actual);

}
```

Figura 38 - Excerto do código do primeiro teste do método getMinNumberOfKeysHigherThanQuantity

O segundo teste serve para verificar se a função devolve os valores que realmente correspondem à realidade.

```
@Test
public void getMinNumberOfKeysHigherThanQuantityTest2() {

    ArrayList<Pair<Area, Integer>> list = smallProduction.getTotalValueList(smallProduction.getProductionPerArea());
    long quantity = 28154945;
    int actual = smallProduction.getMinNumberOfKeysHigherThanQuantity(list, quantity);
    int expected = 0;
    Assert.assertEquals(expected, actual);

    quantity = 21978760;
    actual = smallProduction.getMinNumberOfKeysHigherThanQuantity(list, quantity);
    expected = 2;
    Assert.assertEquals(expected, actual);

    quantity = 21978759;
    actual = smallProduction.getMinNumberOfKeysHigherThanQuantity(list, quantity);
    expected = 1;
    Assert.assertEquals(expected, actual);

    list = production.getTotalValueList(production.getProductionPerArea());
    quantity = 1886178944;
    actual = production.getMinNumberOfKeysHigherThanQuantity(list, quantity);
    expected = 3;
    Assert.assertEquals(expected, actual);

    quantity = 1886178943;
    actual = production.getMinNumberOfKeysHigherThanQuantity(list, quantity);
    expected = 2;
    Assert.assertEquals(expected, actual);

}
```

Figura 39 - Excerto do código do 2º teste ao método getMinNumberOfKeysHigherThanQuantity

Melhoramentos possíveis

Relativamente a possíveis melhorias podemos indicar que é necessário estudar melhor as estruturas de dados existentes para compreender se as estruturas que usamos são as mais adequadas dadas as circunstâncias.

Consideramos ainda que seja possível estruturar melhor a parte da leitura de ficheiros, para ser possível ler ficheiros com diferentes estruturas além das indicadas.

Conclusão

Em suma, neste trabalho aprendemos a distinguir métodos eficientes de organização de dados, de métodos menos eficientes. Aprendemos também a trabalhar de forma mais adequada com os algoritmos que manipulam as estruturas de dados.

Podemos concluir ainda, que quanto mais estudado for o caso e melhor planeado for o trabalho mais facilmente se absorvem as alterações que acontecem ao longo do processo.

Posto isto, consideramos que, apesar de ser possível efetuar algumas melhorias neste trabalho, o projeto encontra-se de acordo com os requisitos pedidos.