Estruturas de informação 2022/2023

4 dezembro

Instituto Superior de Engenharia do Porto

Pedro Teixeira, 1211184
JoãoFernandes,1211681
David Mendonça,1211572
Afonso Machado, 1190326

isep
instituto superior de engenharia do porto

Índice

Contents

Introdução

Este relatório tem o propósito de demostrar os conteúdos que foram abordados e praticados durante a realização do segundo Sprint do Projeto Integrador do 1ºsemestre do 2ºano de Licenciatura em Engenharia Informática.

Relativamente à cadeira de ESINF este Sprint continha User Stories que envolviam o desenvolvimento de funcionalidades que usassem a linguagem JAVA e que permitiriam gerir a informação sobre vários clientes e produtores numa rede de produção biológica e as respetivas encomendas e stock de cada produtor ou Hub. A informação encontrava-se em arquivos de texto com formato CSV.

As funcionalidades incluem o carregamento de um ficheiro contendo todas as encomendas e stock existente, por parte dos clientes e produtores, respetivamente.

Ao longo deste projeto, foram utilizados Graphs e MapGraphs que foram desenvolvidos durante as aulas pratico-laboratoriais tendo utilizado também os algoritmos disponibilizados nas mesmas.

Para cada uma dessas funcionalidades vamos demonstrar o respetivo diagrama de classes, algoritmo utilizados e análise de complexidade.

US 307

Enunciado

- Importar a lista de cabazes

Implementação e análise de complexidade

No Sprint 1 houve a importação de informação acerca de clientes e produtores a partir de um ficheiro CSV. Neste 2ºSprint vamos proceder para a importação tanto das encomendas por parte dos clientes, como o stock disponibilizado pelos produtores, em dados dias.

```
public static Pair<Integer, Integer> importBasketList(File file) {

    FilesReaderApp.lastOrdersFile = file;
    App.getInstance().getCompany().getStock().getStock().clear();

    int clients = 0;
    int producers = 0;

    try {
        Scanner scanner = new Scanner(file);
        String[] line = scanner.nextLine().split( regex: ",");

        ArrayList<String> productsName = new ArrayList<>();
        int len = line.length;

        for (int i = 2; i < len; i++) {
            line[i] = takeCommasOut(line[i]);
            productsName.add(line[i]);
        }

        int lineCounter = 1;
```

Figura 1 – método importBasketList

```
while (scanner.hasNextLine()) {
    line = scanner.nextLine().split( regex: ",");
    if (line.length == len) {
        for (int i = 0; i < line.length; i++) {
            line[i] = takeCommasOut(line[i]);
        }
        ClientsProducers cp = App.getInstance().getClientProducerByCode(line[0]);
        if (cp != null) {
            int day = Integer.parseInt(line[1]);
            ArrayList<Product> products = new ArrayList<>();

            for (int i = 2; i < len; i++) {
                products.add(new Product(productsName.get(i - 2), Float.parseFloat(line[i])));
            }

            ClientBasket basket = new ClientBasket(cp, products);

            if (isThisHub(cp, idx: 0)) {
                App.getInstance().getCompany().getOrders().addHubOrder(day, basket);
            }
```

Figura 2 – continuação figura 1

```
                if (cp.getType().equalsIgnoreCase(Constants.PRODUTOR)) {
                    App.getInstance().getCompany().getStock().addStock(day, basket);
                    producers++;
                } else {
                    App.getInstance().getCompany().getOrders().addOrder(day, basket);
                    clients++;
                }
            } else {
                System.out.println("Client Producer in line " + lineCounter + " not found");
            }
        }
        lineCounter++;
    }

} catch (FileNotFoundException e) {
    System.out.println("File not found");
    return null;
}

return new ImmutablePair<>(clients, producers);
}
```

Figura 3 – fim do método importBasketList

A importação do conteúdo do ficheiro é feita a partir de um método estático (importBasketList) da classe FileReaderApp. O ficheiro é digitado previamente a partir da interface que é apresentada ao utilizador. Cada linha do ficheiro para ser corretamente validade deve seguir a seguinte estrutura:

```
"Clientes-Produtores","Dia","Prod1","Prod2","Prod3","Prod4","Prod5","Prod6","Prod7","Prod8","Prod9","Prod10","Prod11","Prod12"
```

Eventualmente pode existir mais produtos.

Em primeiro lugar o scanner lê uma linha do ficheiro.

De seguida, divide o seu conteúdo a partir de um identificador, que neste caso é a “,”. Se existir aspas num nome, estas são retiradas e adicionadas de volta ao array.

A partir do conteúdo dividido sabemos que a posição 0 do array está identificado o cliente, produtor ou hub.

É retirada da linha o dia que a encomenda ou colocação de stock vai estar presente.

Tendo em conta se é hub/produtor ou cliente, colocamos em diferentes stores que podem ser acedidas na classe App.

Tendo em conta o pior caso possível, a análise de complexidade desta US é a seguinte:

Na figura 1, evidenciamos a presença de um pequeno ciclo for, que tem grau de complexidade O(i-2), sendo i o número de elementos do array. Na figura 2, temos O(i) e O(i-2) na primeira e segunda estrutura “for” respetivamente. Por fim, na figura 3, não existe nada mais no algoritmo que comprometa a análise da complexidade. Concluindo, a complexidade total tendo em conta o pior caso possível é O(3i-4).

Diagrama de Classes

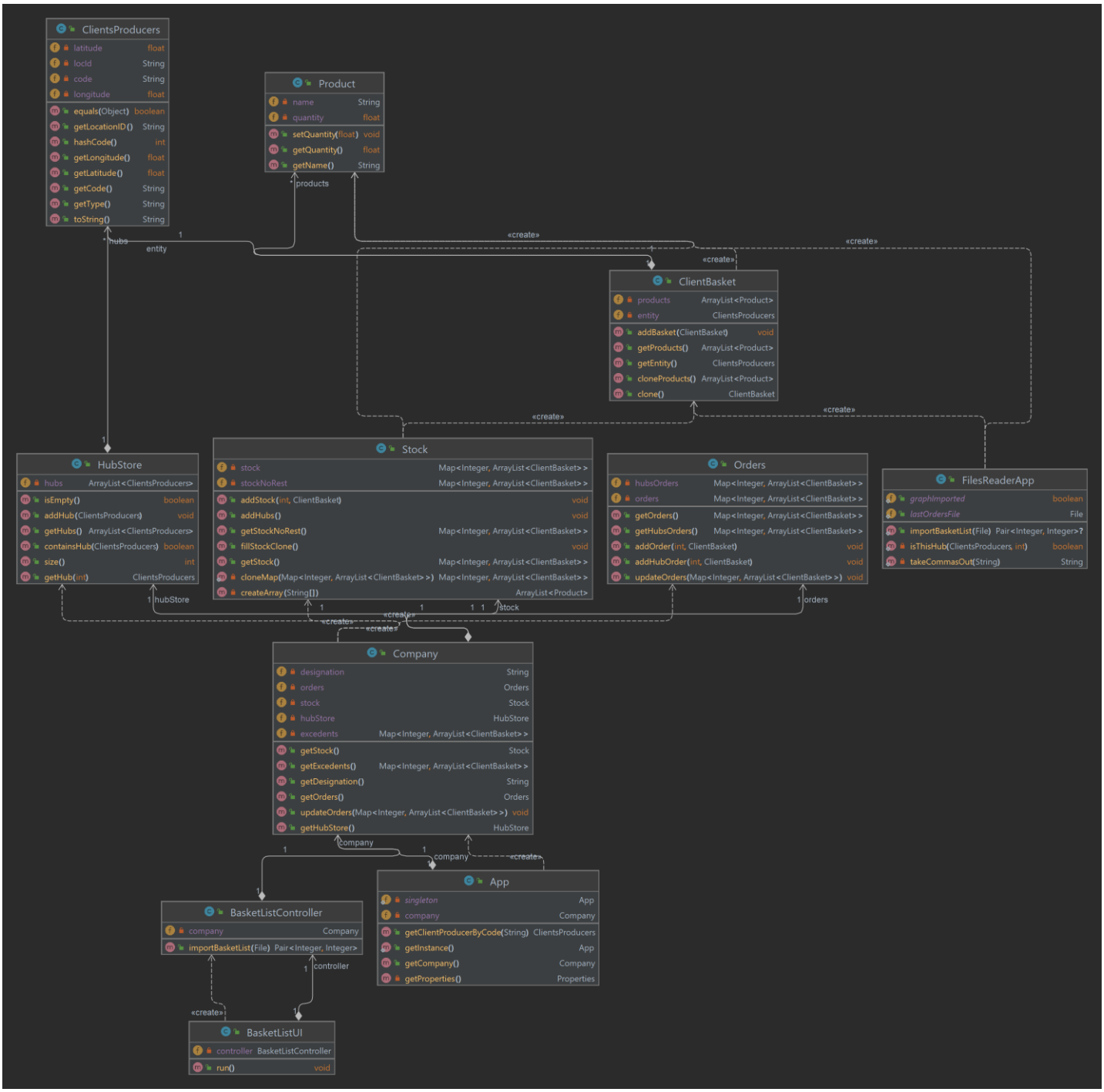

Figura 4 – diagrama de classes US 307

US 308

Enunciado

- Gerar uma lista de expedição para um determinado dia que forneça os cabazes sem qualquer restrição quanto aos produtores.

Implementação e análise de complexidade

Partindo do ponto que previamente foi importado, tanto informação quanto aos clientes e produto, tanta informação sobre a encomendas e stock existente, temos as condições necessárias para criar uma lista de expedição de um certo dia.

Na interface desta funcionalidade é pedido ao utilizador o dia em que pretende saber o que foi expedido na totalidade, ou seja, que clientes foram satisfeitos, e que produtores forneceram os produtos das suas encomendas.

Antes de criar a lista de expedição do dia selecionado, temos antes de tudo de calcular os excedentes de dias, se o dia selecionado não for o primeiro que é apresentado no ficheiro, ou seja, fazer uma espécie de simulação das listas de expedição dos dias anteriores, e assim desta forma também conseguimos saber se os hub (se estes forem definidos previamente) terão produtos expostos para venda, o que simboliza que alguma das encomendas de um dado hub foi correspondida, mesmo que não seja na sua totalidade.

```
public static void CalculateSurplus(int actualDay) {

    App.getInstance().getCompany().getStock().fillStockClone();
    Map<Integer, ArrayList<ClientBasket>> stock = cloneMap(App.getInstance().getCompany().getStock().getStockNoRest());
    Map<Integer, ArrayList<ClientBasket>> orders = cloneMap(App.getInstance().getCompany().getOrders().getOrders());

    if (!stock.isEmpty()) {
        Set<Integer> days = getDays(stock);
        days.removeIf(day -> day >= actualDay);
        if (!days.isEmpty()) {
            Iterator<Integer> iterator = days.iterator();

            CalculateSurplusDay(iterator, stock, orders);

            App.getInstance().getCompany().getExcedents().clear();
            App.getInstance().getCompany().getExcedents().putAll(stock);
        }
    }
}
```

Figura 5 – método CalculateSurplus

Neste excerto, é onde iniciamos o cálculo/simulação dos excedentes de stock. Retiramos da chave os dias que são iguais e maiores que o dia inserido, e enviamos para um método que através de recursividade, dar-nos-á acesso aos excedentes dos dois dias anteriores ao dia que utilizador digitou.

```java
// For each day
if (stock.containsKey(day)) {
    ArrayList<ClientBasket> clientBasketsOrders = orders.get(day);      // orders of day key
    ArrayList<ClientBasket> clientBasketsStock = stock.get(day);        // stock of day key
    ArrayList<ClientBasket> clientBasketsSurplusOlder = new ArrayList<>();
    ArrayList<ClientBasket> clientBasketsSurplusRecent = new ArrayList<>();

    // get surplus of the day before
    if (stock.containsKey(day - 1)) {
        clientBasketsSurplusRecent = new ArrayList<>(stock.get(day - 1));
    }

    newer = !clientBasketsSurplusRecent.isEmpty();

    // get surplus of 2 days before
    if (stock.containsKey(day - 2)) {
        clientBasketsSurplusOlder = new ArrayList<>(stock.get(day - 2));
    }

    older = !clientBasketsSurplusOlder.isEmpty();
```

Figura 6 – início do método CalculateSurplusDay

Esta figura corresponde ao início do método recursivo que tem como objetivo criar os excedentes de stock. Começa por identificar o stock do primeiro dia que estava no keyset que restringimos no método da figura 1, e também se existirem os excedentes do dia anterior e 2 dias anteriores, respetivamente.

```java
// for each client basket in orders
for (ClientBasket clientBasketsOrder : clientBasketsOrders) {

    ArrayList<Product> productsOrder = new ArrayList<>(clientBasketsOrder.getProducts());       // products of one order
    boolean isHub = isThisHub(clientBasketsOrder.getEntity(), idx: 0);
    int hubID = -1;
    if (isHub) {
        hubID = findHubID(stock, clientBasketsOrder.getEntity(), day: day + 1, idx: 0);
    }

    // for each product inside each client basket in order
    // product of one order
    for (Product productOrder : productsOrder) {
        int productID = -1;

        if (productOrder.getQuantity() > 0) {

            if (isHub) {
                productID = findProductID(stock, productOrder, day: day + 1, hubID, idx: 0);
            }
            ClientsProducers producer;
            producer = thereIsProduct(clientBasketsStock, clientBasketsSurplusOlder, clientBasketsSurplusRecent, productOrder, clientBasketsOrder.getEntity());
            int productOwner;
            int productOwnerRecent = -1;
            int productOwnerOlder = -1;
```

Figura 7 – continuação figura 6

```java
private static boolean isThisHub(ClientsProducers entity, int idx) {
    if (idx >= App.getInstance().getCompany().getHubStore().getHubs().size()) return false;

    if (App.getInstance().getCompany().getHubStore().getHub(idx).equals(entity)) {
        return true;
    }
    return isThisHub(entity, idx: idx + 1);
}
```

Figura 8 – método recursivo isThisHub

De seguida, começamos por verificar se quem faz encomenda é um hub, e se for o caso, o booleano isHub fica a true.

Depois vamos ter duas estruturas "for" encadeadas que servem para percorrer em cada encomenda cada um dos produtos encomendados. Para cada produto encomendado verificamos no stock existente e excedentes se existe algum produtor que o tenha, independentemente se consegue corresponder à totalidade do pedido, daí o termo "sem restrições" (que também se dirige a não haver limitações de distância, ou seja qualquer produtor pode servir qualquer cliente).

```java
if (isHub) {
    stock.get(day + 1).get(hubID).getProducts().get(productID).setQuantity(quantity);
    int hID = findHubID(App.getInstance().getCompany().getStock().getStockNoRest(), clientBasketsOrder.getEntity(), day: day + 1, idx: 0);
    int pID = findProductID(App.getInstance().getCompany().getStock().getStockNoRest(), productOrder, day: day + 1, hID, idx: 0);
    App.getInstance().getCompany().getStock().getStockNoRest().get(day + 1).get(hID).getProducts().get(pID).setQuantity(quantity);
}
```

Figura 9 – atualização de stock do dia seguinte de um hub

Depois de atualizado o stock sofrido pela montagem de um cabaz para uma dada encomenda é verificado se o recetor é um hub a partir do booleano que foi alterado ou não no início de cada encomenda, se for, no dia seguinte esse hub vai ter o stock atualizado com aquilo que ele recebeu da encomenda feita.

Tendo em conta o pior caso possível, a análise de complexidade desta US é:

Na figura 8, conseguimos observar que existem duas estruturas "for", logo podemos concluir que inicialmente temos O(n^2), considerando n, o número de encomendas.
Dentro dessas estruturas são chamados os métodos isThisHub (figura 8) e FindHubID que têm a mesma complexidade já que são recursivos e têm praticamente o mesmo percurso, só que retornam objetos diferentes. A complexidade destes dois algoritmos é O(2^n), sendo n o número de hubs na HubStore.

```java
private static int findProductID(Map<Integer, ArrayList<ClientBasket>> stock, Product product, int day, int hubID, int idx) {
    if (idx >= stock.get(day).get(hubID).getProducts().size()) return -1;

    if (stock.get(day).get(hubID).getProducts().get(idx).getName().equalsIgnoreCase(product.getName())) return idx;

    return findProductID(stock, product, day, hubID, idx: idx + 1);
}
```

Figura 10 – método recursivo findProductID

Como acontece anteriormente dentro da segunda estrutura "for" são chamados métodos recursivos com a mesma praticidade dos dois anteriores, só que retornam objetos diferentes, que neste caso são os índices de produtor, e índices de produtos com stock para corresponder a encomendas feitas. A complexidade destes algoritmos também são O(2^n) , tendo em conta que n simboliza o número de produtores e o número de produtos, respetivamente.

Diagrama de classes

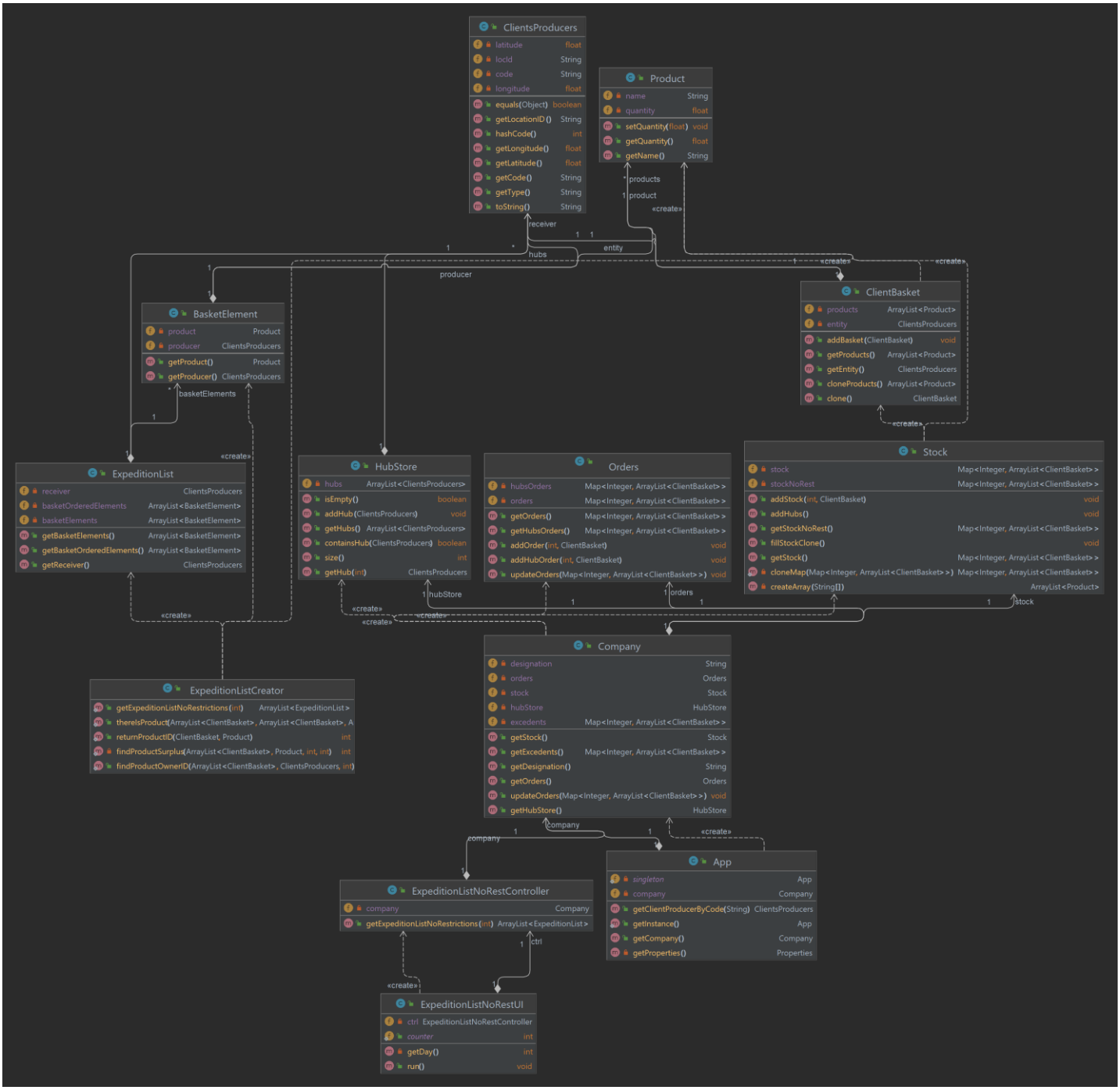

Figura 11 – diagrama de classes US 308