Universidade do Minho
Escola de Engenharia

Laboratórios de Informática III

Maio 2020


Pedro Costa (A85700)


Miguel Carvalho (A84518)


Maria João Moreira (A89540)

Conteúdo

1 Introdução

Mantém-se tudo o que foi dito na introdução do projeto de C. Em suma, continuamos a querer explorar os conhecimentos adquiridos noutras UC's e, a partir dos mesmos, desenvolver estruturadamente uma aplicação de média escala com conceitos que se adequem à dimensão. Mudamos agora a linguagem para o efeito. O paradigma mantém-se dentro de orientado aos objectos, agora numa linguagem que realmente é feita para tal, Java.

Desta vez, iremos tirar partido dos princípios da programação com interfaces que nos irá permitir ter um projeto extremamente flexível, como se fosse um puzzle composto por várias peças. A grande vantagem de tamanha flexibilidade é que podemos trocar peças do puzzle à vontade, desde que tenham o mesmo formato (neste caso, a mesma interface).

2 Módulos e funcionalidades correspondentes

2.1 Packages principais do GestVendasModel - Visão superficial

O módulo principal da nossa aplicação, o **GestVendasModel** foi desenvolvido à volta dos seguintes packages em que cada armazena um módulo independente:

- **Base**: Este package contém definições para as entidades base de todo o sistema, ou seja, produtos, clientes e vendas. Ainda que algumas destas entidades pudessem ser representadas atomicamente, decidimos encapsulá-las numa interface própria.
- **Catalogs**: Neste package estão presentes as representações do Catálogo de Produtos e do Catálogo de Clientes, dois dos módulos principais que compoem o GestVendasModel.
- **Billing**: Package que contém toda a informação relativa ao módulo de dados da faturação. A faturação por sua vez é um módulo de dados que relaciona produtos com informação relativa às suas vendas.
- **Branch**: Package que contém toda a informação relativa ao módulo de dados que representa uma filial. Uma filial, por usa vez, é um módulo de dados que relaciona clientes com os produtos que compram para **uma** dada filial.
- **Configs e Statistics**: Package que armazena módulos de configurações e estatisticas relativas a um GestVendasModel.
- **Parsers**: Neste package define-se as condições de um parser capaz de carregar um GestVendasModel a partir dos ficheiros de produtos, clientes e vendas fornecidos.

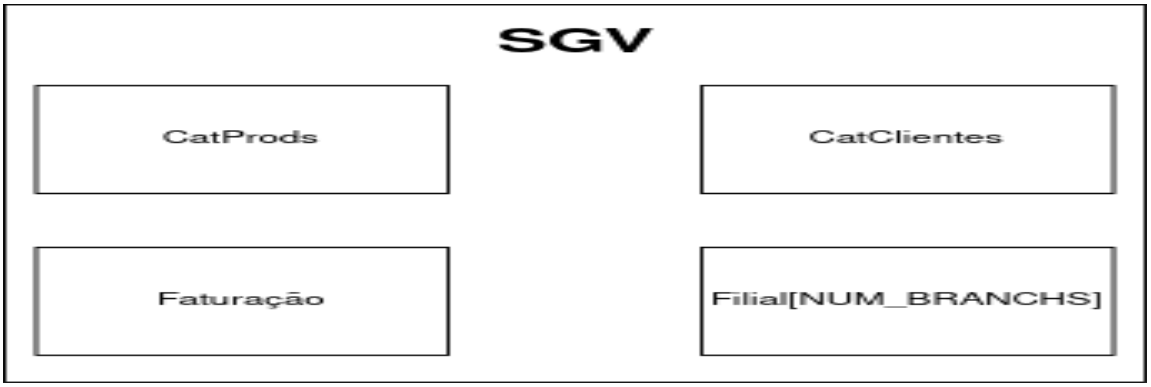

Figura 1: Componentes do SGV

2.1.1 Package Base

Neste package, como explicado anteriormente, guardam-se as interfaces e respetivas implementações das entidades Produto, Cliente e Venda. Ao encapsular os produtos e os clientes (que podiam ser representados atomicamente por uma String) numa interface própria permitimos que, num futuro indeterminado, fosse simples adicionar novos tipos de produtos/clientes mais complexos que não pudessem ser representados atomicamente, sem necessitar de fazer alterações em mais lado nenhum.

2.1.2 Package de Catálogos

Neste package definem-se apenas as interfaces e classes do que são catálogos de produtos e clientes. A interface de ambos é praticamente igual diferindo apenas no facto de armazenar produtos ou clientes.

Permite verificar se existe um produto no catálogo e, obviamente, permite que sejam inseridos no mesmo. Além disso é ainda capaz de nos informar acerca do numero de entradas que contém.

2.1.3 Package Billing

Este package contém todas as interfaces, e respetivas implementações em classes, necessárias para se definir o conceito de Faturação. Como indicado anteriormente, aqui guardam-se relações entre produtos e as suas vendas.

A interface da faturação oferece várias operações distintas começando por permitir faturar vendas, verificar se produtos foram vendidos (e obter a sua informação), obter todos os produtos não vendidos, entre várias outras funcionalidades.

Para esse efeito, encapsulamos todos os dados de vendas dum produto num módulo próprio, sendo esse o **ProductInfo**. Assim, o nosso módulo faturação define apenas um **mapeamento entre produtos e a sua informação**. Tiramos ainda partido de algumas variáveis auxiliares para acumular informação que sabemos que será útil, como se pode ver abaixo.

```
private Map<IProduct, IProductInfo> billing;
private float totalBilled;
private int[][] globalNumberOfSales;
private int number_of_branches;
private int number_of_months;
private int number_of_types;
```

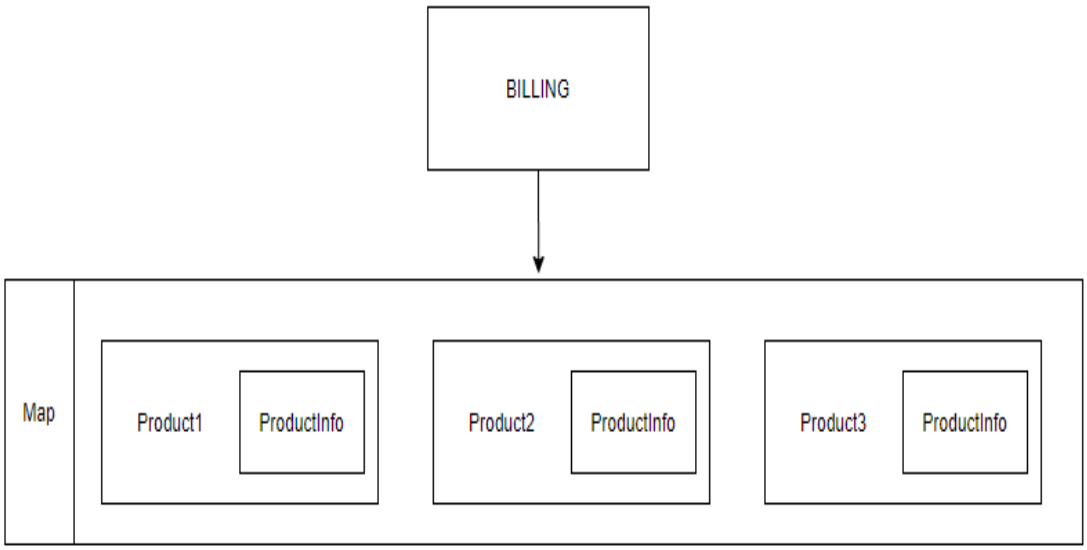

Figura 2: **Esquema do módulo Billing**

2.1.4 Package Branch

Este package contém todas as interfaces, e respetivas implementações em classes, necessárias para se definir o conceito de Filial. Como indicado anteriormente, aqui guardam-se relações entre clientes e os produtos comprados pelos mesmos.

A interface da filial oferece várias operações distintas começando por permitir assinalar que um cliente comprou um determinado produto, verificar quantos compradores distintos houve num determinado mês, obter toda a informação relativa às compras de um cliente, obter o total gasto pelos clientes na filial, entre várias outras operçaões.

Para obter este efeito, encapsulamos todo o historial de compras de cada cliente numa estrutura própria a que chamamos de **ProductsBought**. Aqui temos um mapeamento entre todos os produtos comprados por um determinado cliente e a informação acumulada dos mesmos (encapsulada por sua vez numa outra estrutura apelidada de **CustomerBought**). Tiramos ainda partido de algumas variáveis auxiliares que nos permitiram acumular informação útil numa fase mais avançada, evitando assim procuras desnecessárias mais tarde.

```
private Map<ICustomer, IProductsBought> branch;
private int[] distinctCustomers;
private int number_of_months;
```

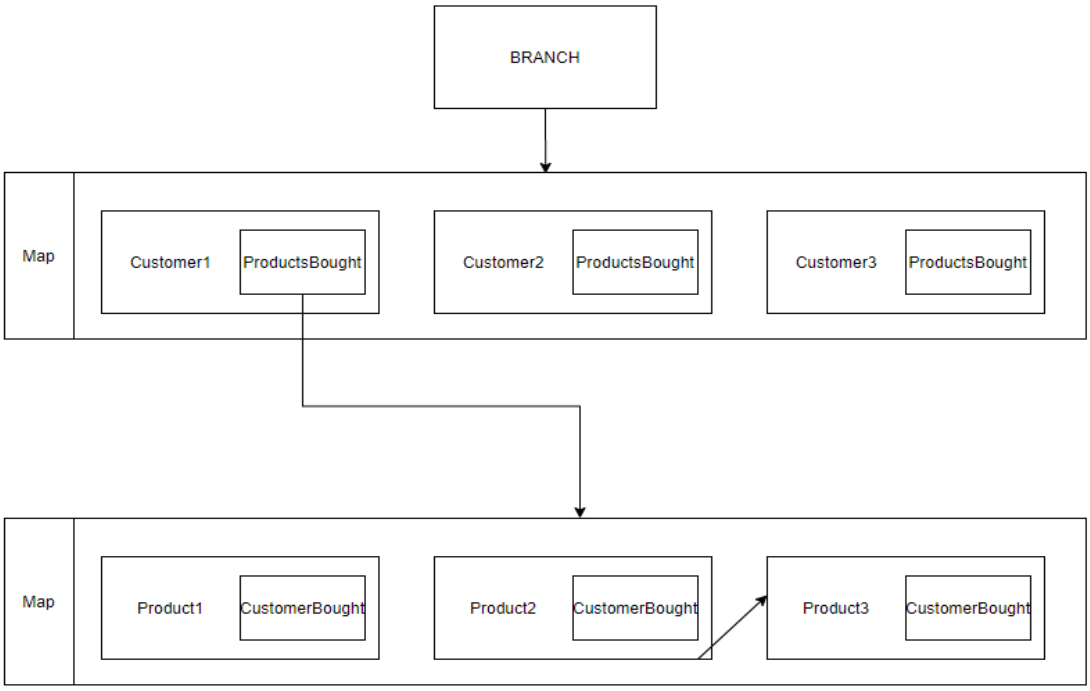

Figura 3: **Esquema de uma Branch**

2.1.5 Package de parsers

Neste package define-se a interface que caracteriza os requisitos mínimos de um parser. A partir daí definimos dois parsers "completos". Um parser que utiliza o package *nio* e lê o ficheiro como uma stream, um outro parser que utiliza um *BufferedReader*.

Além destes, apenas com o propósito de cumprir os testes de performance pedidos, foram desenvolvidos mais seis parsers (três com *streams* e três com *BufferedReader* para fazer parse de todos os ficheiros nos seguintes três cenários:

- **Sem fazer parse**: Apenas percorrer as linhas dos ficheiro de vendas sem fazer absolutamente nada.
- **Sem validação**: Percorrer as linhas dos ficheiro de vendas e construir a estrutura que contém a venda, sem ver se se trata de uma venda válida ou inválida.
- **Com validação e parsing**: Percorrer as linhas dos ficheiros de vendas e construir a estrutura que contém uma venda, assim como ver se esta é válida.

Além desses seis, foram ainda desenvolvidos mais quatro parsers (capazes de cumprir os propósitos acima e também fazer parse normalmente, inserindo nas estruturas de dados as vendas). Estes servem para realizar um dos testes de performance pedidos que se aplica apenas ao ficheiro *Vendas_ 5M.txt* e onde são utilizadas *streams* **paralelas**.

2.1.6 Package de Configs e Statistics

Este package contém as interfaces e respetivas implementações para dois módulos importantes do GestVendasModel.

O primeiro, **Configs**, contém o conjunto de dados iniciais com o qual vamos trabalhar. Isto compreende um conjunto de dados como o número de filiais que temos, onde se localizam os ficheiros que queremos ler, entre vários outros campos.

O segundo, **Statistics**, armazena alguma informação que foi utilizada para responder às queries estatísticas como o número total de vendas, a quantidade de vendas válidas, a quantidade de vendasa com valor zero, entre outros.

2.2 Outros packages desenvolvidos por necessidade/utilidade

2.2.1 Navegador

Este package pertence efetivamente ao conjunto de packages do *model* mas como não é uma peça utilizada diretamente no mesmo não foi abordado acima.

Desta vez houve, em comparação com o projeto anterior, um investimento muito maior no módulo **Navigator** por parte do grupo. Quisemos desenvolver algo estruturado e bem dividido pelo que concluímos que um navegador pode ser abordado em três partes importantes e distintas:

- **Página**: Como seria de esperar, o **Navigator** vai percorrer um conjunto de páginas, cada uma contendo uma porção da informação total que queremos demonstrar. Assim, desenvolvemos uma estrutura que encapsula cada página, o módulo **Page**.
- **Barra de estado**: Um dos problemas que tivemos com este módulo anteriormente foi a falta de comunicação com o utilizador. Assim, definimos uma barra de estado que, consoante o input do utilizador, varia para oferecer comentários e informação útil. Esta funcionalidade foi também encapsulada independentemente numa estrutura que apelidamos de **StatusBar**.
- **Navigator**: Claramente só falta unificar os módulos acima e temos a solução desejada. O **Navigator** é o responsável por guardar um conjunto de várias páginas e uma barra de estado. Define uma API capaz de fazer procuras de expressões regulares ao longo dos dados e navegar sobre as procuras, mudar de página (saltar ou movimentar para a frente ou para trás), dar setup a mensagens de erro para avisar o utilizador quando o input não é reconhecido, entre outras funcionalidades.

O navegador **não dá** display a informação por si só, é apenas um módulo que armazena informação e permite recolher o correspondente à página atual numa determinada estrutura, sendo assim uma componente do *model* e não da *view*.

2.2.2 Package Exceptions

Neste package definem-se várias exceções utilizadas para avisar o utilizador com detalhe do que está a fazer de forma errada, ou desconhecida ao sistema, em qualquer interação.

2.2.3 Package Controller

Neste package encontra-se a interface e a implementação de um controlador que serve de ponte entre o utilizador e a camada de dados, *model*, controlando também o que é mostrado ao utilizador a partir da *view*. Iremos falar com detalhe da arquitetura MVC utilizada brevemente.

2.2.4 Package View

Tal como o anterior, guarda-se aqui a interface e a implementação de uma vista que permite demonstrar informação pré-determinada ao utilizador.

2.2.5 Package Utilities

Aqui guardam-se utilidades de natureza diferente. Temos o módulo **Crono** fornecido pelos professores, temos um módulo **Utilities** que têm apenas o propósito de compor caminhos de maneira a que funcione em Mac e Windows e, temos ainda, uma implementação de um par mutável **MutablePair** que foi extremamente útil numa situação pontual.

2.2.6 Package Testing

Nada neste package é utilizada com o objetivo de desenvolver a aplicação final GestVendasApp. Aqui guarda-se apenas um conjunto de programas externos que fazem *logging* de dados pedidos na secção de medidas de performance.

2.2.7 Encapsulamento

Tal como pedido, garantimos que toda a informação fornecida em qualquer momento é apropriadamente clonada. Assim, garantimos a privacidade dos nossos módulos cuja implementação interna não tem de ser conhecida por ninguém. Além disso, as API's definidas tentam ser tão sucintas quanto possível, evitando deixar fugir detalhes sobre a implementação em qualquer momento.

3 Arquitetura da aplicação

Mais uma vez, e como seria de esperar, utilizamos, para controlar a interação com o utilizador, uma arquitetura MVC em particular. Nesta, é de notar que há uma separação clara de conceitos entre as várias componentes. Eis o papel que cada um assume:

- **Model**: A camada de dados trata apenas de lidar diretamente com dados que recebe ou com os dados que já contém. É capaz de responder a todas as necessidades de um programador que utilize este módulo, de forma genérica. Ou seja, conseguimos devolver todos os conjuntos de dados de forma que depois o programador pode manipular sem grande esforço para obter respostas mais especifícas (nesta secção não vamos ordenar dados por um critério por exemplo).
- **Controller**: O controlador age um pouco, como se pode confirmar na figura abaixo, como um *Man in the middle*. Este vai receber todos os dados do utilizador, passá-los ao *model* para receber a informação que precisa de volta, dar-lhe alguns tweaks caso seja necessário e, de seguida, envia a informação alterada para a *view* para ser demonstrada ao utilizador de uma forma que apenas a própria *view* conhece.
- **View**: A vista tem apenas o objetivo de receber dados de um certo tipo e dar-lhes display de uma forma qualquer. Neste projeto tudo se resume a texto num terminal, mas seria relativamente fácil, devido à modularidade e à arquitetura que temos, migrar as vistas para uma página web, por exemplo.

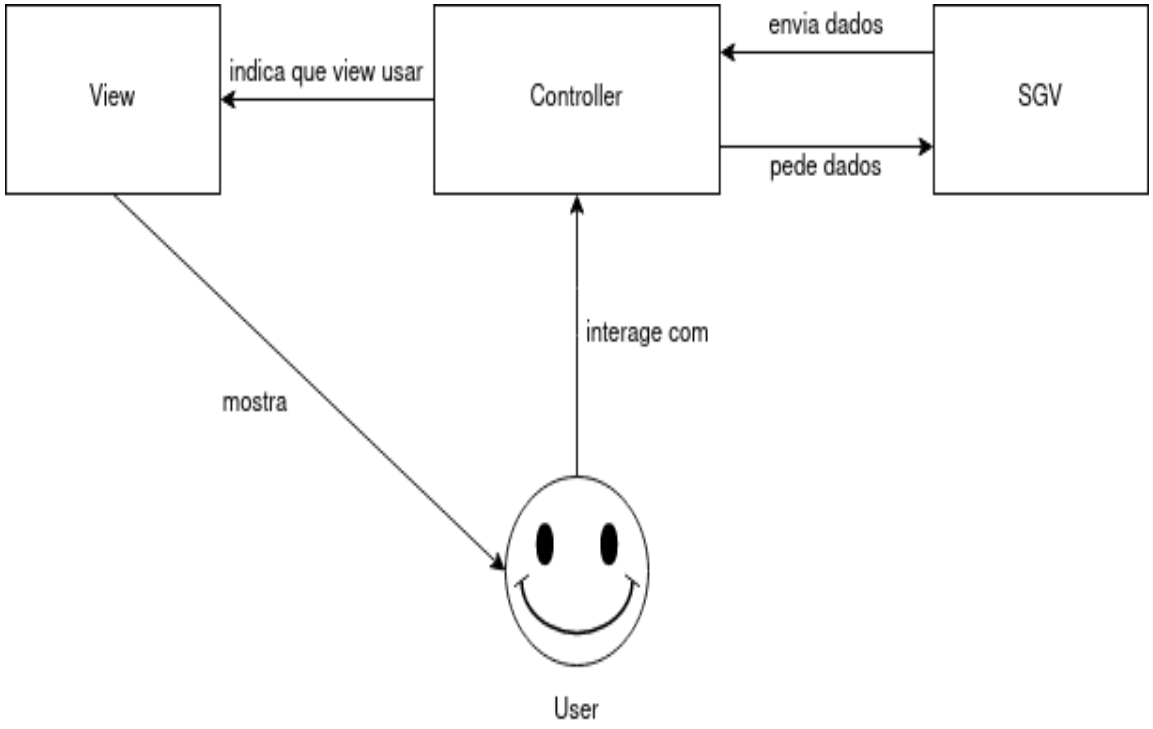

Figura 4: **Design pattern - MVC**

4 Medidas de performance

Um aspeto pedido nesta fase do projeto não pedido anteriormente foi que se fizesse um conjunto de programas auxiliares para medir a performance em alguns cenários. Desenvolvemos os programas auxiliares e obtemos os seguintes resultados.

4.1 Tempo de cada query

Esta não foi uma das medidas pedidas mas tomamos a liberdade de criar um programa externo para extrair estes mesmos dados de forma simples. Eis o obtido.

	Vendas_1M	Vendas_3M	Vendas_5M
Query 1	0.01199022	0.009305349999999999	0.009487769999999998
Query 2	0.01190194	0.007526090000000001	0.01011835
Query 3	0.00104395	1.9472000000000003E-4	1.1379999999999999E-4
Query 4	0.01072582	0.007395110000000001	0.00877747
Query 5	4.0006999999999993E-4	2.3512000000000004E-4	4.5105E-4
Query 6	1.6814085900000002	5.1536569	10.657800149999998
Query 7	0.0057303200000000006	0.007356949999999999	0.007887100000000001
Query 8	0.40231954	1.5416543799999998	3.94023903
Query 9	0.00967734	0.01111796	0.01856353
Query 10	0.41410307	0.6051630100000001	0.8006478599999998

Figura 5: **Tempo médio obtido para cada query ao fim de 10 execuções.**

Nota: O computador utilizado para testes encontrava-se inundado de outras operações em simultâneo pelo que os teus tempos estão muito provavelmente inflacionados.

4.2 Tópicos 1, 2 e 3 das medidas de performance pedidas

Nestes tópicos pedia-se uma análise do tempo de parsing dos ficheiros de vendas utilizando BufferedReader (**BR**) e comparando com o uso do package *nio* que utiliza **streams** para ler o ficheiro. Pretendia-se comparar 3 circunstâncias diferentes, sendo estas:

- **NO_PV**: Sem parsing nem validação (apenas percorrer as linhas).
- **P_NO_V**: Com parsing mas sem validação (percorrer as linhas e criar as estruturas das vendas, sem as validar).
- **PV**: Com parsing e validação.
- **FP**: Full parse, ou seja, parsing, validações e inserções nas estruturas de dados.

Apresentamos então os resultados obtidos num formato de tabela.

	Vendas_1M	Vendas_3M	Vendas_5M
NIO_NO_PV	0.139s	0.426s	0.702s
NIO_P_NO_V	0.474s	1.403s	2.109s
NIO_PV	0.707s	2.145s	3.667s
NIO_FP	2.278s	8.423s	14.746s
BR_NO_PV	0.132s	0.419s	0.666s
BR_P_NO_V	0.425s	1.239s	2.056s
BR_PV	0.684s	2s	3.356s
BR_FP	2.346s	8.267s	14.319s

Como podemos ver pela tabela acima, os tempos crescem (entre **NIO** ou entre **BR**) de forma constante consoante o input aumenta de tamanho. É de notar que o uso de **BufferedReader** obteve praticamente sempre tempos superiores ao uso de **Streams** possivelmente devido a não estarmos a usar um grande volume de dados ou, ainda mais provável, por estarmos a fazer um conjunto bastante pequeno de operações sobre os dados. As streams "benificiam"de muitas operações sobre o conjunto de dados na totalidade.

Apresentamos ainda os resultados obtidos de forma gráfica:

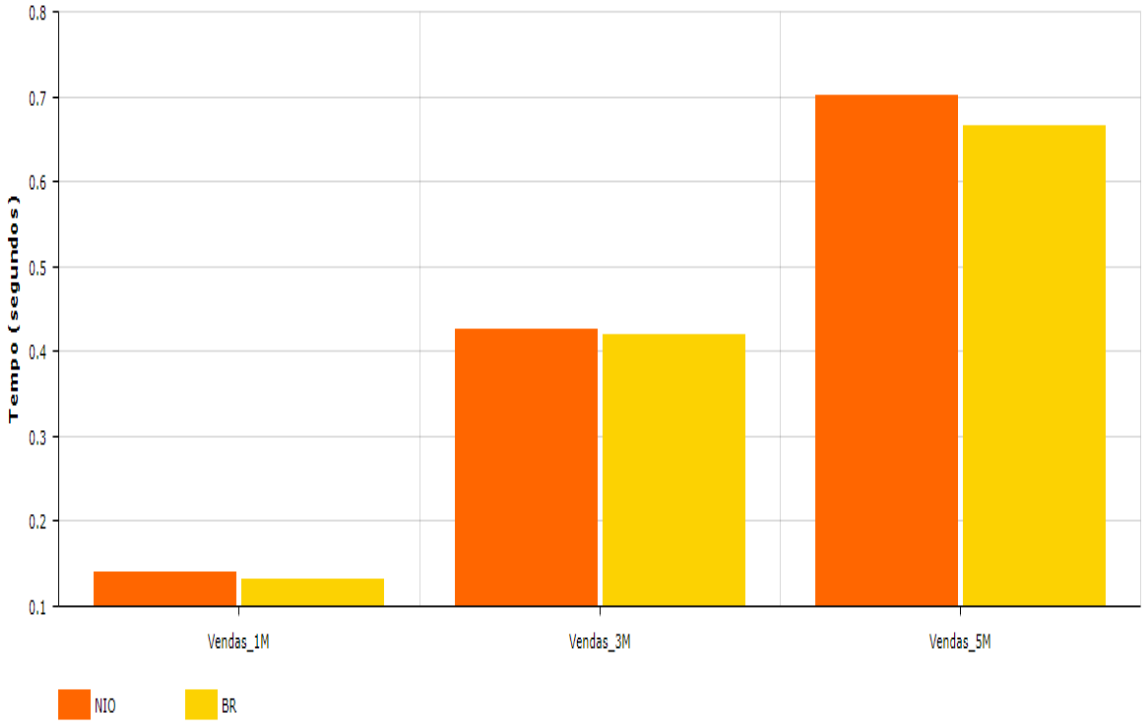

Figura 6: **Resultados obtidos para o cenário sem parsing nem validação**

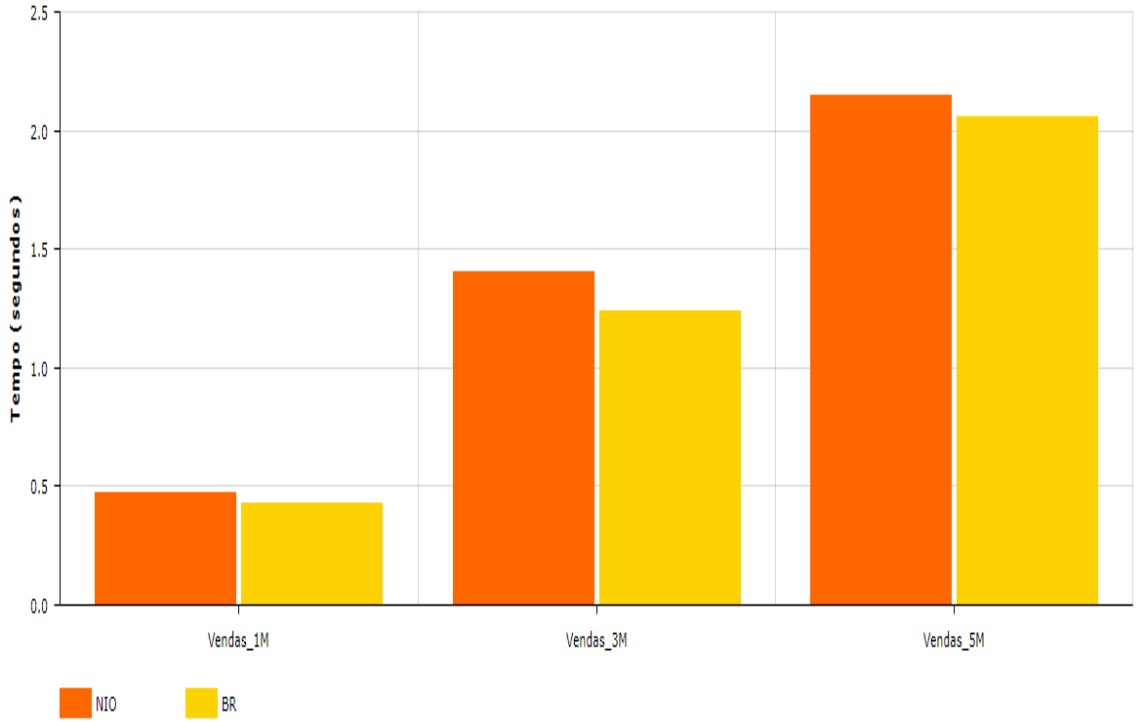

Figura 7: **Resultados obtidos para o cenário com parsing e sem validação**

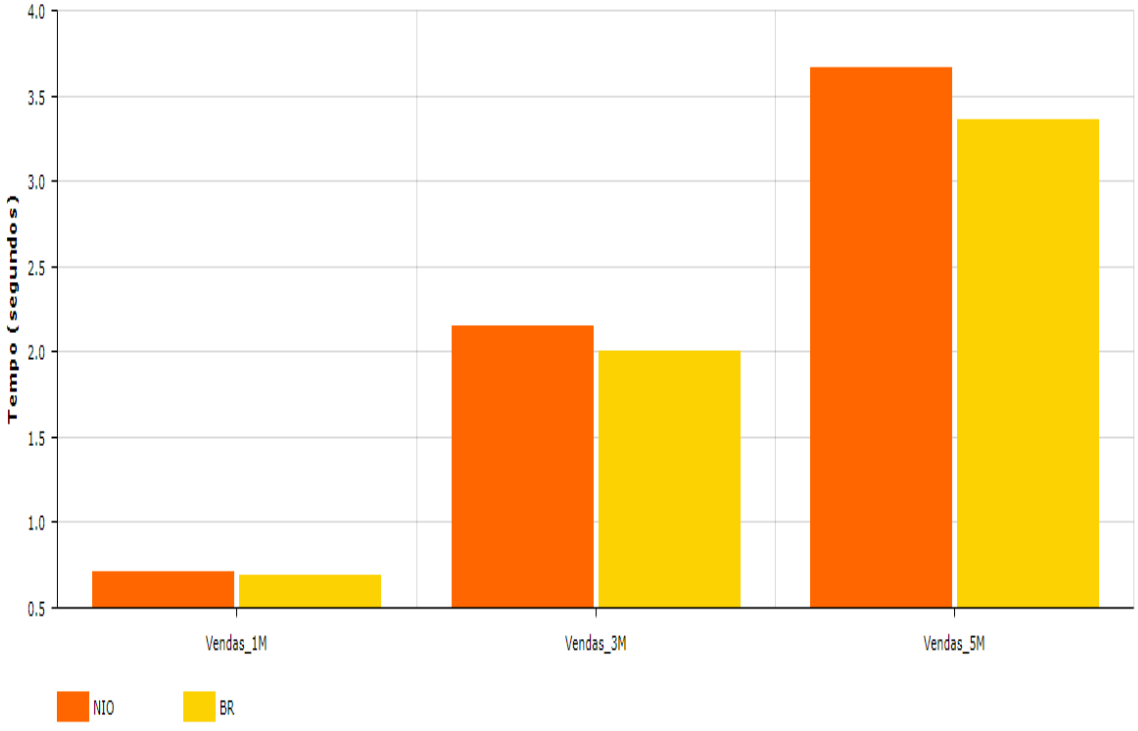

Figura 8: **Resultados obtidos para o cenário com parsing e validação**

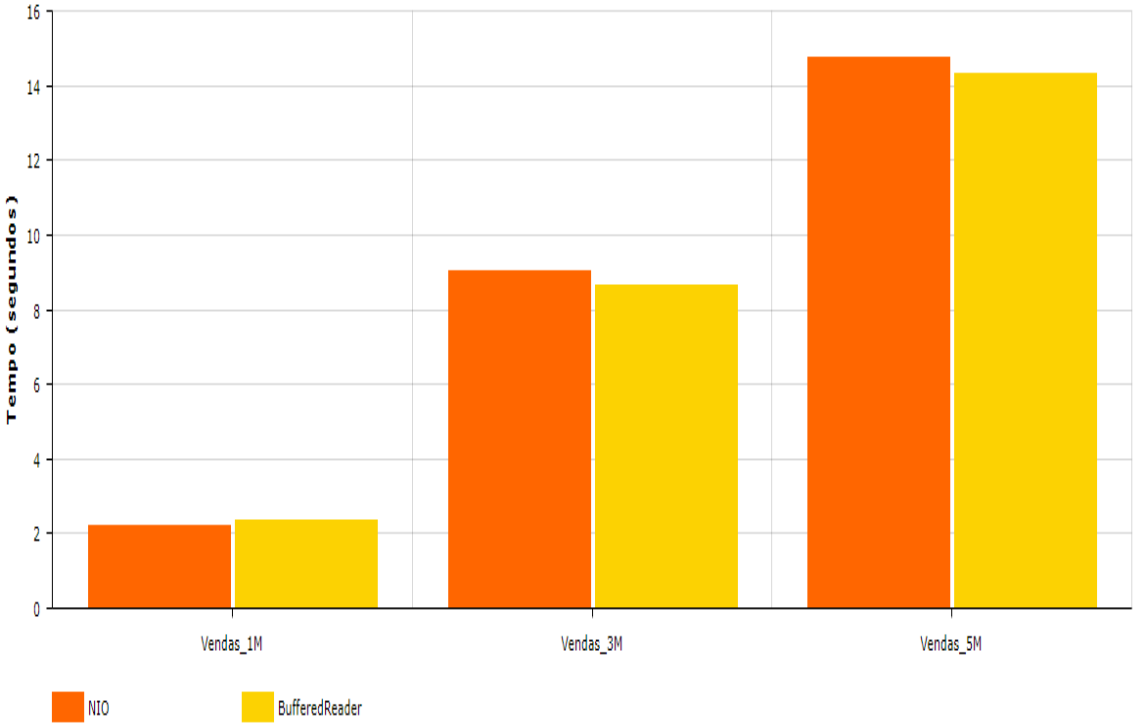

Figura 9: **Resultados obtidos para o cenário com parsing e validação**

4.3 Medições das queries 5 à 9 com implementações diferentes

Neste tópico era pedido que se atuasse sobre as queries 5 à 9. O que se pedia em concreto era apenas que variássemos dentro das possíveis implementações de **Set**, **Map** e **List** consoante o que estivessemos a usar.

No nosso caso, todas essas queries utilizavam apenas um **Map** pelo que alternamos entre implementações com **HashMap** e com **TreeMap**.

	Vendas_1M	Vendas_3M	Vendas_5M
Query 5	0.0004s	0.0006s	0.0007s
Query 6	1.341s	4.012s	7.404s
Query 7	0.007s	0.008s	0.009s
Query 8	0.401s	1.523s	3.763s
Query 9	0.011s	0.012s	0.014s

Legenda: Resultados obtidos com **HashMap**.

	Vendas_1M	Vendas_3M	Vendas_5M
Query 5	0.0006s	0.001s	0.001s
Query 6	1.632s	4.587s	9.871s
Query 7	0.007s	0.010s	0.009s
Query 8 0.403s	s	1.705s	4.064s
Query 9	0.012s	0.012s	0.012s

Legenda: Resultados obtidos com **TreeMap**.

Nota-se então que, para queries mais pesadas, usar o **TreeMap** traz uma grande perda de eficiência. Isto seria expetável uma vez que a vantagem do **TreeMap** é permitir inserções de acordo com um certo critério, algo que não nos importa em nenhuma destas queries.

4.4 Medições relativas a Parallel Streams

Uma outra medição que foi pedida era relativa a fazer parsing do ficheiro de vendas com 5 milhões de linhas utilizando **Streams paralelas**. Consideramos os mesmos cenários da secção **4.2** e um cenário adicional em que fazemos o parsing total (inserindo a informação nas estruturas de dados).

Obtivemos os seguintes resultados:

	NO_PV	P_NO_V	PV	Completo
Vendas_5M	0.961s	1.546s	1.988s	15.833s
Relação em % contra NIO	-27%	36%	84%	-7%
Relação em % contra BR	-31%	32%	68%	-10%

Legenda: Resultados obtidos no parsing do Vendas_5M com **Parallel Streams**.

Nota: Uma percentagem positiva signfica que com parallel streams executou X% mais rápido.

Em comparação com os resultados obtidos com **NIO** e **BufferedReader** para o ficheiro **Vendas_5M** na secção **4.2** nota-se que o cenário **NO_PV** foi bastante mais lento (possivelmente por estarmos a pedir várias threads para fazer algo tão simples). No **P_NO_V** e no **PV** tivemos uma melhoria bastante notável como se pode ver. Surpreendentemente, e

com toda a honestidade não sei como justificar, nota-se que com parallel streams demoramos mais a fazer o parsing total do ficheiro em relação aos outros modos utilizados.

5 Profiling de memória

Para fazer profiling a nível de memória utilizamos a ferramenta **VisualVM** como um plugin do IntelliJ.

6 Utilização do programa e melhorias a fazer

A primeira melhoria óbvia que nos ocorre que podia ser feita era redimensionamento do navegador para se adequar melhor ao tamanho do terminal. À exceção disso acreditamos que temos um trabalho bastante sólido. É muito provável que se for analisado em detalhe se notassem pequenos pormenores que se pudessem melhorar.

A nível de utilização, oferecemos a possibilidade ao utilizador de passar um ficheiro de *Configs* para serem utilizadas como primeiro argumento. Caso nada seja utilizado, recorre-se a configurações default.

7 Conclusão

Este relatório não é final, ainda irá provavelmente haver algum refactoring do texto e será adicionada a informação que resta, nomeadamente, profiling de memória. Cumpre, no entanto, os requisitos minímos e aborda todos os tópicos necessários para a avaliação total do projeto.

O projeto foi desenvolvido no IntelliJ e testado maioritariamente no IDE por isso se houver essa possibilidade, há mais garantias de nenhuma funcionalidade falhar ao correr lá. Devido ao aviso extremamente tardio da necessidade de compilar num JAR apenas conseguimos experimentar em Windows/Linux e funciona relativamente bem. Em windows não conseguimos tirar proveito das pesquisas a cores no navegador pelo menos não na linha de comandos antiga. Em Linux funciona tal como no intelliJ. Recomenda-se apenas a utilização do programa em fullscreen, de outra forma, todas as tabelas serão desformatadas. Mais uma vez, isto deve-se ao facto de ter sido desenvolvido dentro do IDE e esperamos não receber uma penalização por um aspeto como este.

O diagrama de classes está incluido na pasta docs do projeto.

Concluindo, é interessante comparar duas linguagens a implementar o mesmo paradigma. Nota-se que C foi bastante mais eficiente (na versão feita para speed pelo menos) mas Java acaba por tornar mais simples desenvolver uma arquitetura MVC consistente e modularizar tudo com base nas interfaces.

Foi ainda interessante ver que