

UNIVERSIDADE DO MINHO

MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA

LABORATÓRIOS DE INFORMÁTICA III

Sistema de Gestão de Vendas

José Luís (a75481)
Rui Azevedo (a80789)
31 de Maio de 2020

Conteúdo

1	Introdução	5
2	Análise e Especificação	6
2.1	Descrição do problema	6
2.2	Especificação de requisitos	6
3	Arquitectura	7
3.1	Arquitectura da máquina	7
3.2	Arquitectura do sistema	7
3.2.1	Diagrama de classes	8
4	Concepção/Desenho da resolução	9
4.1	Leitura de dados e <i>Parsing</i>	9
4.2	Interfaces	11
4.3	Produtos e Clientes	12
4.4	Model	12
4.4.1	Catálogos	12
4.4.2	Faturação	13
4.4.3	Filial	14
4.4.4	SGV	16
4.5	Persistência dos dados	16
4.6	View	17
4.6.1	MainMenuView	17
4.6.2	ErrorView	17
4.6.3	RequestView	17
4.6.4	TableView	18
4.7	Controller	18
4.7.1	Navegador	19

5	Testes e desempenho	20
5.1	Tempos de execução	21
5.2	<i>Performance</i> de <i>CPU</i> e memória	23
6	C vs. Java	24
7	Conclusão	25
8	Anexos	26

Lista de Figuras

1	Diagrama de classes do sistema	8
2	Diagrama de classes do teste de <i>parsing</i>	9
3	Gráfico dos tempos de execução do <i>parser</i>	11
4	Códigos de produtos (esquerda) e clientes (direita)	13
5	Estrutura de facturação	14
6	Estrutura da filial	15
7	Vista tabular de uma <i>query</i>	18
8	Estrutura de uma página	20
9	Análise de <i>performance</i>	23
10	Diagrama de classes completo	26

Lista de Tabelas

1	Testes de <i>performance</i> de <i>parsing</i>	10
2	Tempo de carregamento de catálogos com diferentes estruturas	12
3	Tempos de execução	21
4	Comparação entre Maps	22

1 Introdução

O presente relatório tem como objectivo apresentar o trabalho desenvolvido na cadeira de **Laboratórios de Informática III**, trabalho esse que pretende dar continuidade ao desenvolvido na fase passada. O objectivo do trabalho é o mesmo do que o da fase anterior mas, desta vez, o trabalho vai ser desenvolvido na linguagem de programação **Java**.

Pretende-se, com isto, obter uma comparação, a nível de desempenho e de implementação, entre as duas linguagens de programação. Contudo, nesta fase, vai-se tomar partido das *Collections* do *Java* e do seu nível de abstracção para experimentar diferentes estruturas de dados para armazenar os dados.

É também pretendido que a aplicação siga uma arquitectura *MVC*, tal como aconteceu na fase anterior. É de esperar que esta arquitectura seja mais fácil de implementar numa linguagem orientada aos objectos do que numa linguagem imperativa.

Numa primeira fase irá ser analisado o problema em causa, definindo a sua arquitectura. De seguida serão apresentadas, detalhadamente, cada uma das componentes que definem a arquitectura *MVC* da aplicação. Serão apresentados também os resultados obtidos pelo programa para cada uma das *queries* definidas bem como os tempos de leitura e carregamento dos ficheiros. Por fim, fazer-se-á uma análise crítica entre os dois sistemas desenvolvidos.

2 Análise e Especificação

Nesta secção irá ser apresentada, de uma forma breve, o objectivo desta fase do trabalho, bem como os desafios impostos para a sua realização.

2.1 Descrição do problema

É pretendido desenvolver um sistema de gestão de vendas, desta vez implementado na linguagem de programação *Java*. Os dados do sistema mantêm-se os mesmos, existem clientes que fazem compras de produtos em filiais, daí haver uma relação entre clientes e produtos. Dado que existe esta relação, é necessário também obter informação sobre a facturação, particular e global, desses produtos, sendo também necessário saber o número de compras e vendas dos mesmos. Com isto, é perceptível que seja necessário criar módulos independentes que armazenem e gerem este tipo de informação.

2.2 Especificação de requisitos

Nesta fase foram impostos novos desafios a nível da implementação. Para além da necessidade de haver uma boa estruturação de dados, é também pretendido que haja uma análise prévia de que classes usar para a leitura dos dados.

A nível da leitura de dados dos ficheiros vão ser comparadas duas classes já integradas na linguagem e são as seguintes:

- *IO* : *package* introduzido pelo *Java 1.0*, orientado a *streams* e a operação de *I/O* é bloqueante. É uma solução simples que não acrescenta muita complexidade ao sistema.
- *NIO* : *package* introduzido no *Java 1.4*, orientado a *buffers* e a operação de *I/O* é não bloqueante. Pode tornar a leitura dos dados mais complexa.

Foi imposto também que sejam analisadas as diferentes colecções do *Java* e que seja feita uma comparação a nível de *performance* ao usar as mesmas alternativamente. Uma vez que o projecto, a nível de estruturas, vai ser baseado em colecções *Map<K,V>*, *Set<E>* e *List<E>*, a análise estrutural vai ter como base a substituição das estruturas *HashMap<K,V>* onde se estava a usar *TreeMap<K,V>* e vice-versa, *HashSet<E>* onde se estava a usar *TreeSet<E>* e vice-versa, e usar *Vector<E>* onde se estava a usar *ArrayList<E>* e vice-versa. Com isto, ir á-se ter uma noção evidente de que estruturas de dados são mais eficientes para o armazenamento e manipulação dos dados.

Relativamente às interrogações, é necessário armazenar um conjunto de dados estatísticos para que, a qualquer altura, o utilizador possa os consultar. Estes dados são, por exemplo, o último ficheiro de vendas lido, número total de registos de vendas errados, número total de produtos, número total de compras por mês, *etc.*

Quanto à arquitectura do sistema, vai-se manter a arquitectura *Model-View-Controller* (MVC), pois permite uma boa modulação do sistema, partindo as diferentes camadas da aplicação em três partes, tal como o nome indica.

Por fim, foi proposto um conjunto de interrogações para serem respondidas através do desenvolvimento de métodos. Sempre que possível, tirou-se partido das *Streams* do *Java* pois permitem uma boa estruturação de uma resposta, tornando o código de certa maneira mais funcional.

3 Arquitectura

Nesta secção irá ser apresentada a arquitectura da máquina onde foram medidos os testes de desempenho da aplicação bem como a arquitectura, mais detalhada, do sistema.

3.1 Arquitectura da máquina

A medição dos testes de desempenho foram realizados num computador *Mac* com as seguintes características:

- Sistema operativo: *OSX* versão 10.14.6
- Processador: *Intel Core i5* 2.3GHz
- Memória: 8GB 2133 MHz LPDDR3

3.2 Arquitectura do sistema

Como já foi referido anteriormente, um dos requisitos a nível arquitectural é a implementação da arquitectura *Model-View-Controller*. Para isso foram criados três *packages*, cada uma referente a cada uma das camadas aplicacionais.

- *Model* : contém toda a lógica de negócio do sistema. Esta camada contém as classes referentes aos catálogos de clientes e produtos, facturação e filiais, bem como as respetivas *interfaces*. A classe principal deste *package* é a classe que agrega toda a informação,

funciona como o *facade* do modelo, pois toda a comunicação com classes externas a este *package* é feita através da *interface* da mesma.

- *View* : camada aplicacional que contém toda a lógica de apresentação de dados ao utilizador. A informação é fornecida pela camada *Controller*, não tendo acesso à camada aplicacional de negócio.
- *Controller* : camada mediadora entre as camadas de visualização e de negócio. O *controller* recebe pedidos do cliente e, mediante o pedido, acede ao *facade* da camada de negócio, que irá tentar obter uma resposta ao pedido. Esta resposta é enviada para o controlador que, por sua vez, irá enviar os dados para camada de visualização para posterior apresentação ao utilizador.

3.2.1 Diagrama de classes

Para uma melhor percepção do sistema que foi desenvolvido, é apresentado de seguida o diagrama de classes da aplicação. É de notar que os métodos e as variáveis de instância foram omitidos bem como alguma notação usual deste tipo de diagramas pois o objectivo é apenas mostrar a relação entre as diferentes componentes do sistema. Foram também omitidas algumas classes desenvolvidas sendo que só estão apresentadas as classes principais. Em anexo, é possível ver o diagrama completo do *model* da aplicação.

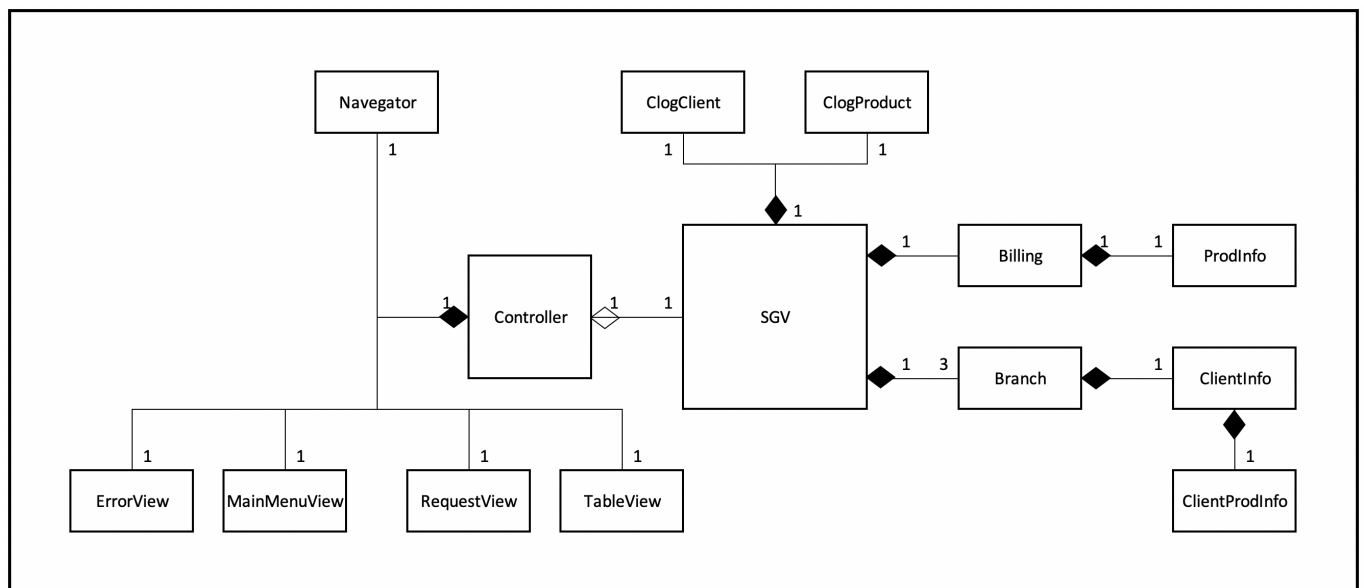


Figura 1: Diagrama de classes do sistema

4 Concepção/Desenho da resolução

De seguida, irão ser apresentadas, mais detalhadamente, as componentes do sistema desenvolvido. Numa primeira fase serão analisados os tempos de carregamento usando distintas bibliotecas do *Java* com o intuito de saber qual a mais eficiente para ser usada, efectivamente, no projecto. Após esta análise, irá ser descrito com mais rigor cada uma das camadas aplicacionais da arquitectura *MVC*. Por fim, irá ser apresentado o método de gravar e carregar os estados das estruturas através de ficheiros objecto.

4.1 Leitura de dados e *Parsing*

Para testar os diferentes tipos de *parsing* aos ficheiros de entrada, foi criado um *package* designado por *ParserTest* que contém as classes usadas para este teste. Tirou-se partido do paradigma da linguagem para criar uma classe abstracta, *Parser*, que contém informação em comum aos dois diferentes tipos de *parsing* feitos, nomeadamente, o caminho para o ficheiro de vendas a ser lido, o conjunto de clientes e produtos, definido como um *Set*, e ainda uma lista de vendas, que irão ser geradas no momento de leitura.

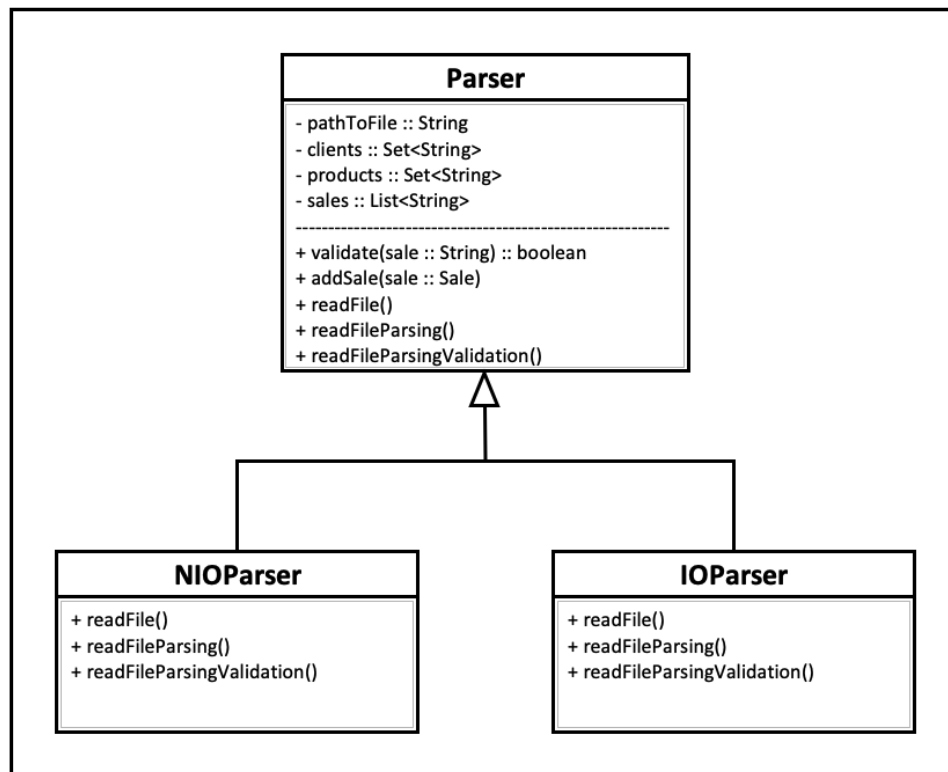


Figura 2: Diagrama de classes do teste de *parsing*

A classe abstracta contém três métodos abstractos, ou seja, cujas sub-classes são obrigadas a implementar:

- *readFile* : método de leitura sem *parsing* e sem validação.
- *readFileParsing* : método de leitura com *parsing* mas sem validação.
- *readFileParsingValidation* : método de leitura com *parsing* e validação.

A classe *Parser* contém também um método de validação de uma venda bem como um método para adicionar uma venda à lista de vendas pois será usado pelas sub-classes implementadas.

Após a implementação deste módulo, os resultados obtidos para os tempos de *parsing*, em segundos, são os apresentados na tabela abaixo. Os valores obtidos são resultantes da média de 10 execuções de cada critério para cada ficheiro.

	Critério	Vendas_1M	Vendas_3M	Vendas_5M
IO Parser	Sem parsing Sem validação	0.1195	0.3215	0.4507
	Com Parsing Sem validação	0.6659	2.3489	4.1962
	Com parsing Com validação	0.9997	3.4474	6.5182
NIO Parser	Sem parsing Sem validação	0.1703	0.3789	0.6465
	Com Parsing Sem validação	0.9091	4.4570	4.3224
	Com parsing Com validação	1.0132	5.6516	5.6920

Tabela 1: Testes de *performance* de *parsing*

Pode-se observar que os módulos do *IO Parser* são mais eficientes em todos os critérios há excepção de um. Quando o ficheiro de vendas tem 5 milhões de entradas e é necessário fazer o *parsing* com validação, os módulos do *NIO Parser* são mais eficientes 0.8263s. No entanto, este aspecto não é suficiente para optar por escolher o *NIO* pois, para isso, seria necessário

realizar testes para ficheiros ainda maiores para tirar a dúvida se esta classe é mais eficiente para ficheiros maiores

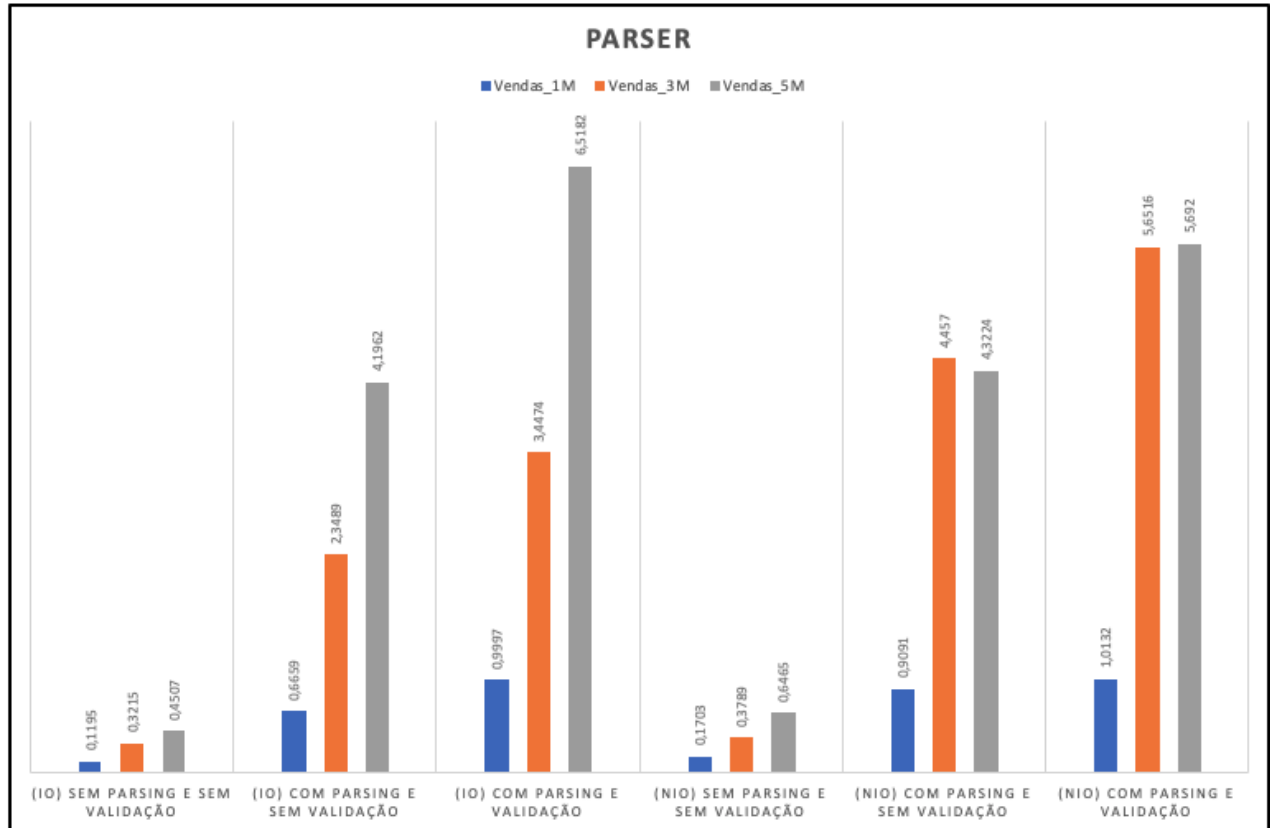


Figura 3: Gráfico dos tempos de execução do *parser*

4.2 Interfaces

As *interfaces* permitem definir o comportamento que uma dada classe de objectos deve ter. Este método de programação torna um sistema mais formal pois define a iteração que uma classe tem com o mundo exterior. A partir do momento que uma classe implemente uma *interface*, é obrigada a implementar todos os métodos nela definidos.

Para tirar o máximo partido disto, todas as classes que pretendem usar outras, o tipo de dados usado deve ser o da *interface* e não o da classe em si. Isto garante que, mesmo que um módulo tenha que ser trocado por outro, esse novo módulo terá que implementar todos os métodos definidos na *interface*, garantindo assim o correto funcionamento do sistema mesmo que haja alterações.

4.3 Produtos e Clientes

Embora os produtos e clientes sejam formados apenas por um código, foram criadas classes específicas para cada um destes elementos. Isto torna o sistema mais versátil se for necessário acrescentar informação a esses elementos. As *interfaces* são, designadamente, ***IClient*** e ***IProduct***.

4.4 Model

Nesta secção irá ser apresentada, em detalhe, a camada de negócio do sistema. Esta camada está dividida em três *packages* principais: catálogo, facturação e filial. Dentro de cada *package* estão definidas as classes e *interfaces* para o correto funcionamento da aplicação.

4.4.1 Catálogos

Para a estruturação da classe dos catálogos, foram experimentadas duas abordagens. A primeira foi criar uma estrutura semelhante à implementada na primeira fase do trabalho. O catálogo era baseado numa lista de árvores balanceadas, onde havia uma função de *hash* que, através da parte alfabética de um código, indicava a árvore que continha os códigos dos produtos/clientes que começavam por esse conjunto de letras. A segunda abordagem, passou por simplificar esta estrutura e defini-la apenas como uma colecção do *Java*, mais propriamente, um *Set<String>*, que guarda os códigos de clientes/produtos.

De seguida, pode-se observar a diferença dos tempos de carregamento, em segundos, de clientes e produtos.

	List<SortedSet<String>>	HashSet<String>	TreeSet<String>
Carregamento	2.3811	1.8189	2.4339

Tabela 2: Tempo de carregamento de catálogos com diferentes estruturas

Pode-se observar que a estrutura mais eficiente para carregamento de catálogos é o *HashSet<String>* uma vez que a complexidade de inserção é praticamente constante. Os catálogos de clientes e produtos são então, respectivamente, definidos como *Set<IClient>* e *Set<IProduct>*.

Tanto a classe de catálogo de produtos como a de clientes partilham a mesma *interface* e que obriga a implementar os seguintes métodos:

- `getSize` : devolver o número de elementos da estrutura.
- `isValid` : verificar se um código de produto é válido
- `contains` : verificar se um dado elemento pertence à estrutura
- `addElem` : adiciona um elemento à estrutura

Para que a *interface* permita servir as duas classes, esta usa um tipo genérico de dados sendo o protótipo da *interface* o seguinte: ***public interface ICatalog<T>***, onde todos os métodos definidos na *interface* trabalham sobre esse tipo genérico de dados.

A função de validação de códigos de cada uma das classes permite que haja alguma flexibilidade caso seja necessário alterar a estrutura dos mesmos. A função baseia-se no número de letras que compõem um código (*depth*), na gama de valores que as letras podem tomar (*range*) e também na gama de valores que a parte numérica pode tomar (*minRange* e *maxRange*).

Se for necessário alterar a estrutura de um código, é possível fazê-lo através do ficheiro de configurações, que irá ser abordado em secções posteriores, e manter o funcionamento correto do programa. No entanto, isto só é possível se o código for composto por um conjunto de letras seguido de um conjunto de dígitos.

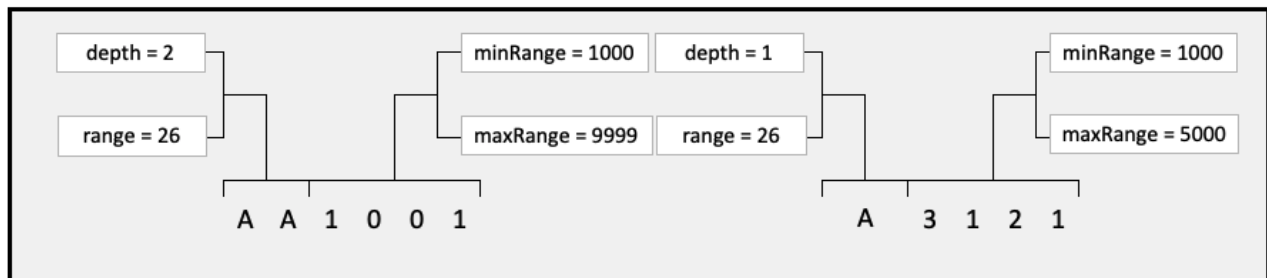


Figura 4: Códigos de produtos (esquerda) e clientes (direita)

4.4.2 Faturação

Este módulo tem como objectivo guardar toda a informação relativa à venda de um produto, não contendo qualquer referência a clientes.

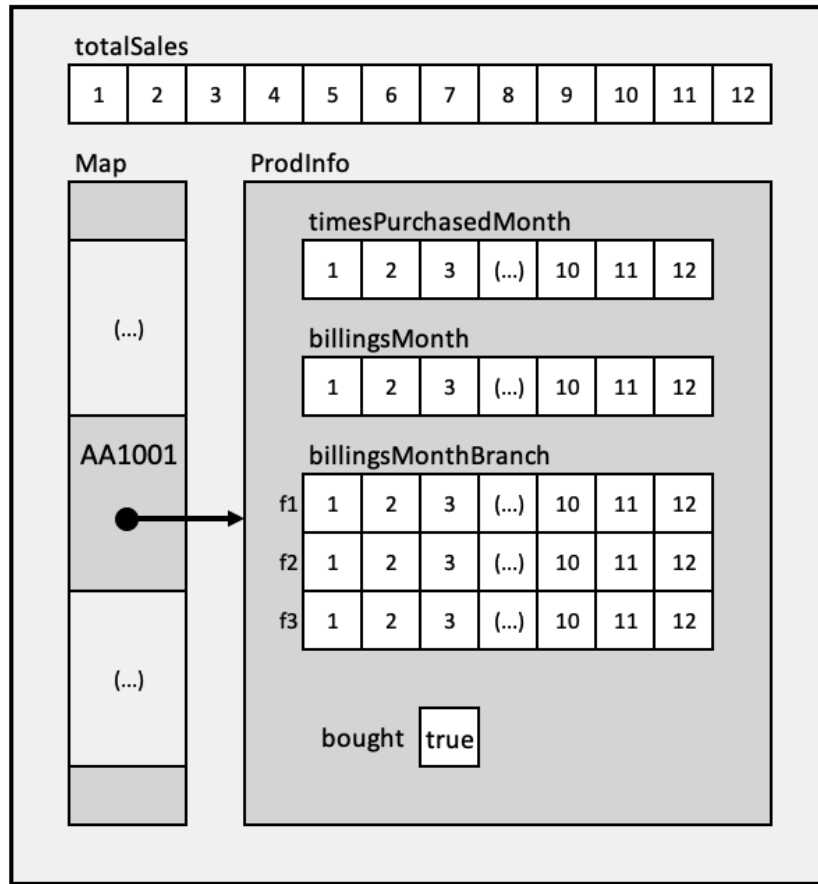


Figura 5: Estrutura de facturação

É composto então por um $Map<IProduct, IProdInfo>$, estrutura que guarda a correspondência entre os produtos do sistema e a sua informação, e mais uma variável referente a dados globais, nomeadamente, o número de vendas por mês. A classe *ProdInfo* contém quatro variáveis de instância: um valor booleano que representa se esse produto foi comprado ou não, o número de vezes que esse produto foi comprado em cada mês, o total facturado por esse produto em cada mês e, por fim, para cada filial, a facturação por mês.

4.4.3 Filial

A gestão de filial apresenta a estrutura mais complexa do sistema, tal como aconteceu na última fase do trabalho.

Este módulo contém a relação entre clientes e respectivos produtos comprados, representada com um $Map<IClient, IClientInfo>$. Para além disso, contém dados globais, nomeadamente, o número de clientes que têm registos de compras, o número de vendas numa deter-

minada filial, o número de clientes distintos que fizeram compras em cada mês e o número de vendas por mês.

A classe *ClientInfo* guarda a correspondência de todos os produtos comprados por um determinado cliente e a informação relativa a esse produto, guardada num *Map<IProduct,IClientProdInfo*. É também composta pelo número de compras por mês de um cliente, pelo dinheiro gasto por mês e pelo número de unidades compradas.

Por fim, a classe *ClientProdInfo* contém a informação específica de um determinado produto comprado por um determinado cliente. É composta pelo número de unidades compradas desse produto, o número de vezes que foi comprado, globalmente e por mês, e o facturado com esse produto, globalmente e por mês.

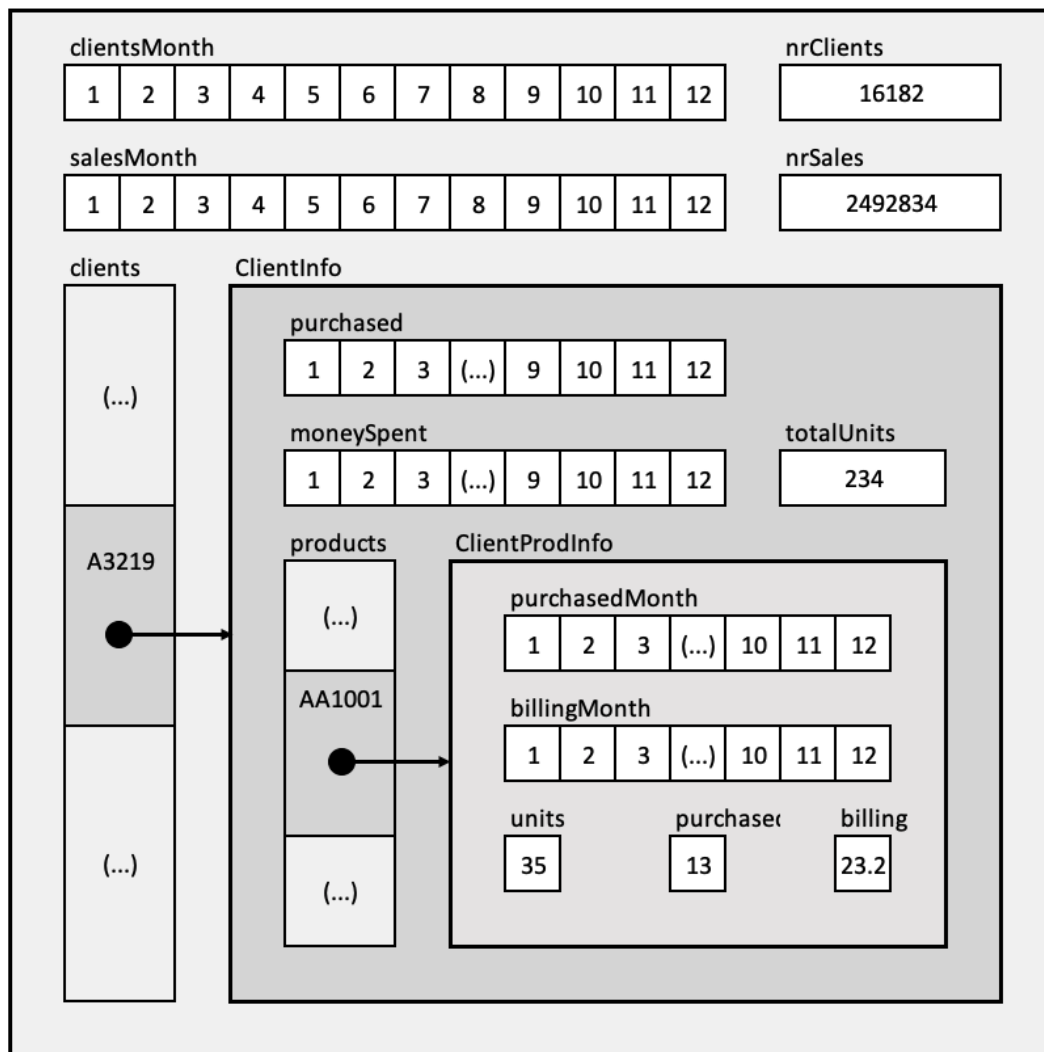


Figura 6: Estrutura da filial

4.4.4 SGV

A classe agregadora de toda a informação é a classe *SGV*. Pode ser vista como o *facade* da camada de negócios pois, é através da sua *interface*, que existe comunicação externa com outras camadas. A classe é responsável por carregar as estruturas de dados, bem como dar resposta às *queries* propostas para o trabalho. As suas variáveis de instância são:

- *ICatalog<IClient>* : catálogo de clientes
- *ICatalog<IProduct>* : catálogo de produtos
- *IBilling* : módulo de faturação
- *List<IBranch>* : lista com os módulos de filial
- *Configuration* : módulo de configuração das constantes do sistema. O formato do ficheiro de configurações é *.properties* onde é permitido adicionar as variáveis na forma *var = value*, com a vantagem de não ser precisa uma ordem de escrita das variáveis. Quando o sistema arranca, o ficheiro *config.properties* é lido e esta estrutura é carregada com dados para posterior acesso.
- *Stats*: módulo que guarda a informação de dados estatísticos do sistema, referentes ao último ficheiro de vendas lido assim como referentes ao estado atual do sistema.

O módulo agregador contém, então, toda a lógica de acesso às diferentes componentes que compõem a camada de negócio da aplicação, bem como lógica de obter uma resposta para cada *query* proposta.

4.5 Persistência dos dados

A qualquer momento, o utilizador pode guardar o estado do sistema para um ficheiro de objectos para posterior leitura. A classe agregadora do programa contém funções para gravar e carregar a estrutura geral do sistema através de um ficheiro dado como *input*. Para isso, foram usados os *ObjectStreams* do *Java* que permitem esta funcionalidade de uma maneira bastante simples, desde que se garanta que todas as classes que vão ser persistidas implementem a *interface Serializable*.

A nível de leitura de dados, esta funcionalidade tem uma complexidade bem maior do que ler directamente dos ficheiros de dados. Isto deve-se à complexidade das estruturas de dados do sistema, pois os *ObjectStreams* têm que guardar, não só os dados mas como todas as referências entre estruturas o que torna o processo bastante dispendioso.

O tempo de guardar o estado é de 24.1590s e o de carregamento é de 30.8064s, o que é bastante superior ao tempo de *parsing* anteriormente apresentado. Pode-se concluir então que, neste caso, não compensa usar ficheiros objectos dada a complexidade que traz ao sistema de carregamento.

4.6 View

Para esta camada do sistema foram criadas quatro classes para representação visual dos dados. Cada *view* contém a sua própria *interface* para comunicação com o *controller*. Para a representação visual de tabelas foi usada uma dependência designada por *FlipTables* que permite, de uma forma muito intuitiva, criar *pretty tables* para apresentação dos dados.

4.6.1 MainMenuView

Vista responsável pela impressão do menu principal da aplicação. Esse menu contém a informação de todas as funcionalidades do sistema.

4.6.2 ErrorView

Esta vista é bastante simples pois a sua única funcionalidade é imprimir situações de erro. O método definido nesta classe tem como parâmetro a mensagem de erro que se pretende imprimir. Após a impressão da mensagem, é feito um *sleep* de três segundos para o utilizador ter tempo de ler a mensagem de erro antes de o ecrã limpar.

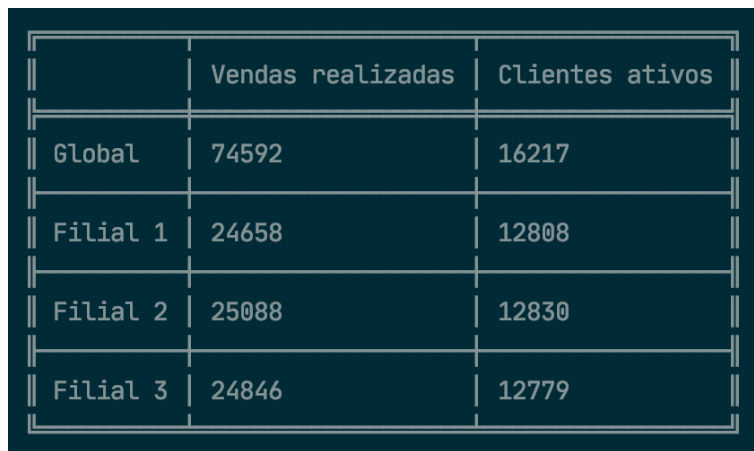
4.6.3 RequestView

A *requestView* contém métodos utilitários que serviram de auxílio no desenvolvimento do controlador do sistema. Esta classe contém os seguintes métodos:

- *clearScreen* : limpa o ecrã imprimindo um conjunto de linhas em branco
- *showTime* : apresenta uma mensagem própria para apresentar o tempo de execução de uma *query*
- *printMessage* : imprime uma mensagem ao utilizador
- *printQueryHeader* : imprime o cabeçalho de uma *query*

4.6.4 TableView

Esta classe tem como objectivo apresentar conteúdo em formato tabular. Para a representação dos dados das *queries* que têm formato tabular, usa-se o método *simpleTable* que recebe como parâmetros, os cabeçalhos superiores de uma tabela, opcionalmente, os cabeçalhos laterais e por fim um *array* bidimensional de objectos relativos ao conteúdo da *query*. Para a representação de dados por página, foi definida a função *printTable* que recebe os dados necessários para a impressão da mesma. Para além dos métodos de apresentação dos dados das *queries*, apresenta também métodos para imprimir, em forma tabular, resultados estatísticos, recebendo como parâmetro uma cópia da classe *Stats*.



	Vendas realizadas	Clientes ativos
Global	74592	16217
Filial 1	24658	12808
Filial 2	25088	12830
Filial 3	24846	12779

Figura 7: Vista tabular de uma *query*

4.7 Controller

Esta camada aplicacional serve de intermediária entre a camada de negócio e a camada de apresentação. O controlador é responsável por ler os dados fornecidos pelo utilizador, processar esses dados, comunicar com a *interface* do modelo a fim de obter uma resposta para o cliente e, por fim, enviar os dados à camada de apresentação para a sua representação.

Esta classe usa uma classe auxiliar de conversores do tipo obtido pelas diferentes interações para o tipo de dados a ser usado pelas *views* respectivas. Desta maneira, e com as *views* desenvolvidas para a aplicação, o fluxo de cada *query* foi feito de uma maneira bastante sistemática. Pode-se apresentar o fluxo de uma *query* da seguinte maneira:

Algorithm 1 Fluxo das *queries*

```
1: function FLUXO
2:
3:   if sgvsLoaded() then
4:     args = readArgs() (opcional)
5:     Time.start()
6:     result = query(args)
7:     time = Time.stop()
8:     content = converter(result)
9:     printResult(content, time)
10:  else
11:    printError()
12:
```

4.7.1 Navegador

A classe navegador pertence à camada do controlador. É aqui que é definida a lógica de paginação e iteração sobre os resultados de uma *query* que precise deste tipo de apresentação.

Esta classe é composta por diversos componentes, sendo os mais importantes:

- List<List<Object>> data : contém a informação a ser apresentada onde, cada lista dentro da lista corresponde a uma coluna de uma tabela.
- List<Object> title : estrutura para guardar o título de cada tabela. Usada essencialmente na última *query* devido à sua estrutura.
- String[] topHeaders : cabeçalhos superiores da tabela de uma página.
- String[] headers : cabeçalhos laterais da tabela de uma página (opcionais).
- currentPage : página atual que se está a consultar.
- maxPage: número máximo de páginas
- pageSize: número de elementos por página.

A imagem abaixo apresenta uma vista de uma página. Pode-se ver a relação que existe entre os componentes da classe e a respectiva vista.

Tempo: 0.037015445s

Cliente	Valor gasto
J3651	88748,5
J4911	27914,88
D4190	120046,5
A2582	73334,1
J1135	1271,28

(A) Página anterior | (D) Página seguinte | (/field:valor) Procura por elemento | (número) Número da página
 Página 1/1
 <\$> |

Figura 8: Estrutura de uma página

Para iterar sobre as páginas, as seguintes acções são possíveis com este módulo:

- **A** : imprime a página anterior
- **D** : imprime a página seguinte
- **Número** : Vai para a página com o um determinado número
- **/campo:valor** : É possível, dado um campo da tabela (número da coluna) e um valor, ir para a primeira página que faz *match* com esta informação. Por exemplo, numa *query* que imprima produtos e o seu total facturado (duas colunas na tabela), o comando /2:123,2 apresentava a primeira página onde aparece o preço 123.2, caso exista. A validação dos campos é feita através de uma expressão regular com o seguinte formato:

$$\backslash s^* \backslash d^+ \backslash s^* : \backslash s^* \backslash w^+ (, \backslash d^+) ? \backslash s^*$$

5 Testes e desempenho

Nesta secção serão apresentados os tempos de execução de cada *query* bem como a análise dos seus tempos. Para além disso, irá ser feita uma breve análise sobre a *performance* do *CPU* e da memória quando o programa está em execução.

5.1 Tempos de execução

Para testar a aplicação desenvolvida, foi criado um módulo à parte do programa cujo objectivo é executar todas as *queries* e medir os tempos de execução. Os valores apresentados de seguida dizem respeito a uma média de cinco execuções do programa e o tempo apresentado é em segundos. As estruturas de dados usadas para este teste foram *HashSets* para os catálogos e *HashMaps* para todas as outras estruturas, pois foram as que apresentaram melhores tempos, como se irá ver de seguida.

	Vendas_1M	Vendas_3M	Vendas_5M
Parsing	2.1966	6.3217	11.4072
Query 1	0.0139	0.010	0.0081
Query 2	0.0154	0.0186	0.0181
Query 3	0.0013	0.0016	0.0017
Query 4	0.0161	0.0194	0.01954
Query 5	0.0010	0.0019	0.0021
Query 6	0.9774	4.0346	9.0613
Query 7	0.0399	0.0491	0.0540
Query 8	0.3322	1.0460	3.4950
Query 9	0.0146	0.0199	0.0182
Query 10	0.5196	0.5829	1.9904

Tabela 3: Tempos de execução

A nível de tempos de carregamento, e, comparando os valores obtidos com os do programa desenvolvido em *C*, pode-se observar que estão bastante melhores. Os tempos medidos no *parsing* da primeira fase do trabalho para os três ficheiros foram, respectivamente, 4.0455s, 13.7661s e 23.7561s. É notável que, com as estruturas usadas, o *parsing* do ficheiro de 5 milhões de vendas é mais rápido que o *parsing* do ficheiro de 3 milhões de vendas do primeiro trabalho. Isto deve-se ao facto de se estar a usar *HashSets* e *HashMaps* para armazenar todos os dados, garantindo um tempo de consulta e inserção praticamente constante, enquanto no projecto de *C* estava-se a usar um *array* de árvores binárias o que torna a consulta e inserção logarítmica.

Quanto às *queries*, os tempos obtidos foram bastante razoáveis, havendo apenas três *que-*

ries que ultrapassam 1 segundo de execução no ficheiro de 5 milhões de vendas. Todas as *queries*, à excepção da 6, 8 e 10, apresentam um tempo praticamente constante entre ficheiros pois os dados a agregar conseguem-se obter quase só com os dados contidos nas estruturas de dados. No entanto, as *queries* 6, 8 e 10, apresentam maior complexidade de agregação de dados e, mesmo que essas *queries* peçam um limite de valores a apresentar, é sempre necessário calcular os valores totais para posteriormente extrair um limite de valores.

Para ter uma melhor percepção de qual a melhor estrutura a usar para o trabalho, foram feitos testes a outras estruturas. O primeiro teste a ser feito foi trocar a implementação dos catálogos de *HashSets* para *TreeSets* mantendo tudo resto inalterado. Esta troca aumentam o tempo de *parsing* em 5s por isso, podemos concluir que os *HashSets* têm um melhor desempenho. O próximo teste a fazer foi trocar todos os *HashMaps* para *TreeMaps*. Como se pode ver na tabela abaixo, os tempos de *parsing* dispararam drasticamente com esta mudança, sendo mais de o dobro o tempo para carregar os dados. Quanto às *queries*, estas apresentam também, relativamente, melhor tempo. Dado isto, para a representação das estruturas usaram-se *HashMaps*.

(Vendas_5M)	TreeMap	HashMap
Parsing	26.8799	11.2340
Query 1	0.0106	0.0078
Query 2	0.0183	0.0164
Query 3	0.0021	0.0016
Query 4	0.0051	0.0198
Query 5	0.0032	0.0024
Query 6	9.0384	8.6112
Query 7	0.0473	0.0536
Query 8	4.4293	3.4934
Query 9	0.0503	0.0178
Query 10	2.1791	1.9845

Tabela 4: Comparação entre Maps

5.2 Performance de CPU e memória

Para a análise da utilização de *CPU* bem como a memória que está a ser usada, foi usado a aplicação *Java VisualVM* que permite visualizar valores gerais de execução.

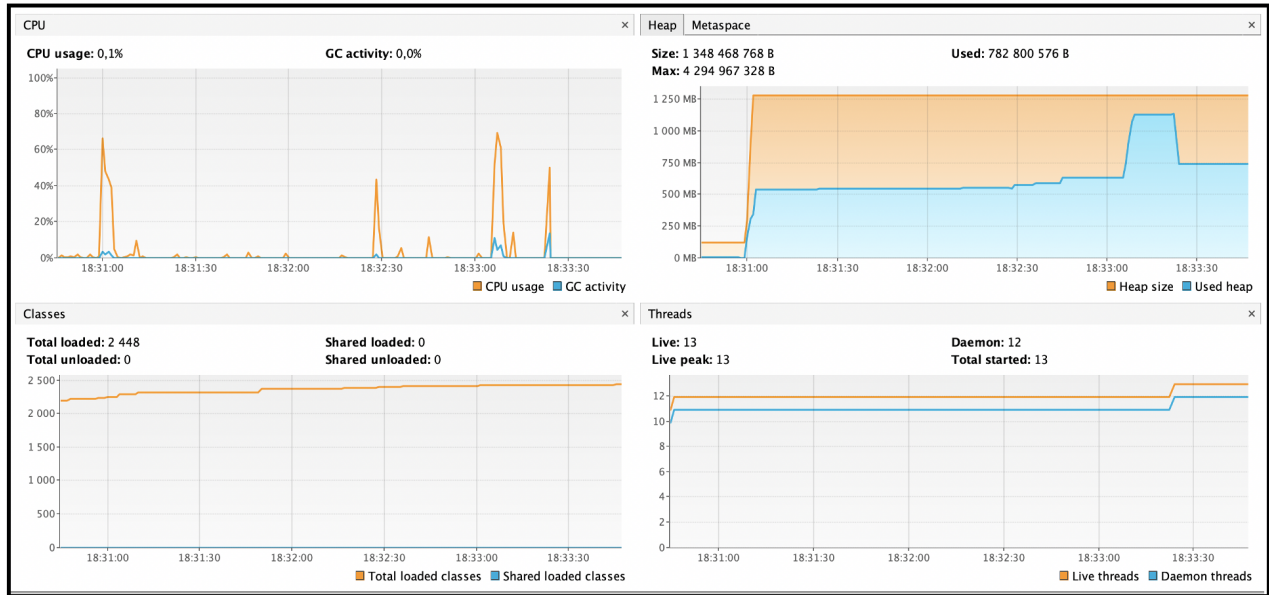


Figura 9: Análise de *performance*

A informação acima diz respeito à execução de todas as *queries* do sistema incluindo *parsing*.

O primeiro pico é referente ao *parsing* do ficheiro de 1 milhão de vendas. Era de esperar que esta operação tivesse uma utilização relevante do *CPU*, neste caso 60% de utilização. Como era de esperar também, e uma vez analisados os tempos de execução das *queries*, pode-se ver pelos picos que as *queries* 6, 8 e 10 apresentam maior utilização de *CPU*, especialmente a *query* 10 que está ao mesmo nível do *parsing*. O último pico na utilização de *CPU* diz respeito ao *garbage collection* efetuado manualmente através da *interface* da *Java VisualVM*.

Quanto à memória, o primeiro pico é, analogamente, referente ao *parsing* e, nesse momento as estruturas ficaram carregadas com os dados e, por isso, a utilização da *heap* aumenta para 500MB. Durante a execução das *queries* a utilização da *heap* vai aumentando mas quase linearmente. O segundo pico que aparece é referente à execução da *query* 10, que dobra o valor do tamanho utilizado. É de notar que, caso o comando de *garbage collection* não fosse executado, a *heap* mantinha 1000 MB de utilização. No entanto, mesmo com a execução do *garbage collection*, a memória utilizada passa para os 750MB, e não para os 250MB relativos ao início do programa.

6 C vs. Java

Fazer praticamente o mesmo trabalho nestas duas linguagens de programação deu para comparar as diferenças entre as mesmas. O tempo de desenvolvimento do sistema usando *Java* foi bem mais curto do que o usado para a primeira fase do trabalho. Isto deve-se, essencialmente, ao facto do *Java* dispor de um conjunto de colecções capazes de armazenar dados de forma eficiente, enquanto em *C*, se não for usada nenhuma biblioteca como o *glib*, não é possível. Para além disso, a quantidade de problemas que surgiram foram muito menos do que os de *C*, isto muito provavelmente devido à gestão de memória que o *Java* faz automaticamente.

A nível de desempenho, uma vez que as *queries* a desenvolver eram diferentes daquelas implementadas na primeira fase, o ponto comum é a leitura de dados. Nesta segunda fase, os tempos de carregamento foram melhores devido à estrutura usada para armazenar os dados. Todos os módulos da camada de negócio usam *Maps* para armazenar dados o que torna os tempos de procura e adição melhores do que usar um *array* de árvores binárias.

Outra vantagem do *Java* é a facilidade de implementar abstracção de dados. Usando as *interfaces* definidas foi muito fácil testar várias variantes de estruturas de dados para o modelo da aplicação.

A vantagem do *C* em relação ao *Java*, que por sua vez também pode ser uma desvantagem, é o controlo que se tem sobre a memória. Com as estruturas de dados adequadas, é possível obter um desempenho, provavelmente, melhor do que o do *Java*, mas requer mais esforço na gestão de memória. Como se pode observar na análise de memória, por exemplo, quando se executou a *query 10*, pode-se observar que a *heap* não ficou com o tamanho utilizado igual ao tamanho que tinha quando as estruturas foram carregadas, o que era desejável. É provável que em *C* fosse mais fácil ter este controlo.

7 Conclusão

Este trabalho foi um dos mais interessantes de toda a licenciatura pois obriga a ter uma boa capacidade de estruturação das diferentes componentes do sistema para que o mesmo funcione coerente e eficientemente.

É de elevada importância o desenvolvimento de módulos independentes que comuniquem apenas por uma *interface* pois torna o sistema muito mais funcional e pronto para alterações, uma vez que, dado que existem *interfaces*, a troca de módulos não afecta o correcto funcionamento do programa, desde que essa mesma *interface* seja respeitada.

A arquitectura *MVC* tem um papel fundamental no que foi referido anteriormente. Para este tipo de trabalhos, esta separação de módulos faz bastante sentido e permite ter uma visão estruturada da aplicação como um todo.

A análise de complexidade das diferentes estruturas de dados, bem como das diferentes classes do *Java* foi bastante enriquecedora para conhecer melhor os prós e contras da linguagem, saber que estruturas usar e em que caso, bem como componentes mais eficientes para resolver um determinado problema.

Por fim, o trabalho foi bastante elucidativo e permitiu aumentar os conhecimentos em ambas as linguagens, ficando a saber algumas das suas limitações e das suas valências. Permitiu também aumentar a capacidade de estruturar um problema aparentemente complexo, em pequenas partes menos complexas que possam coexistir.

8 Anexos

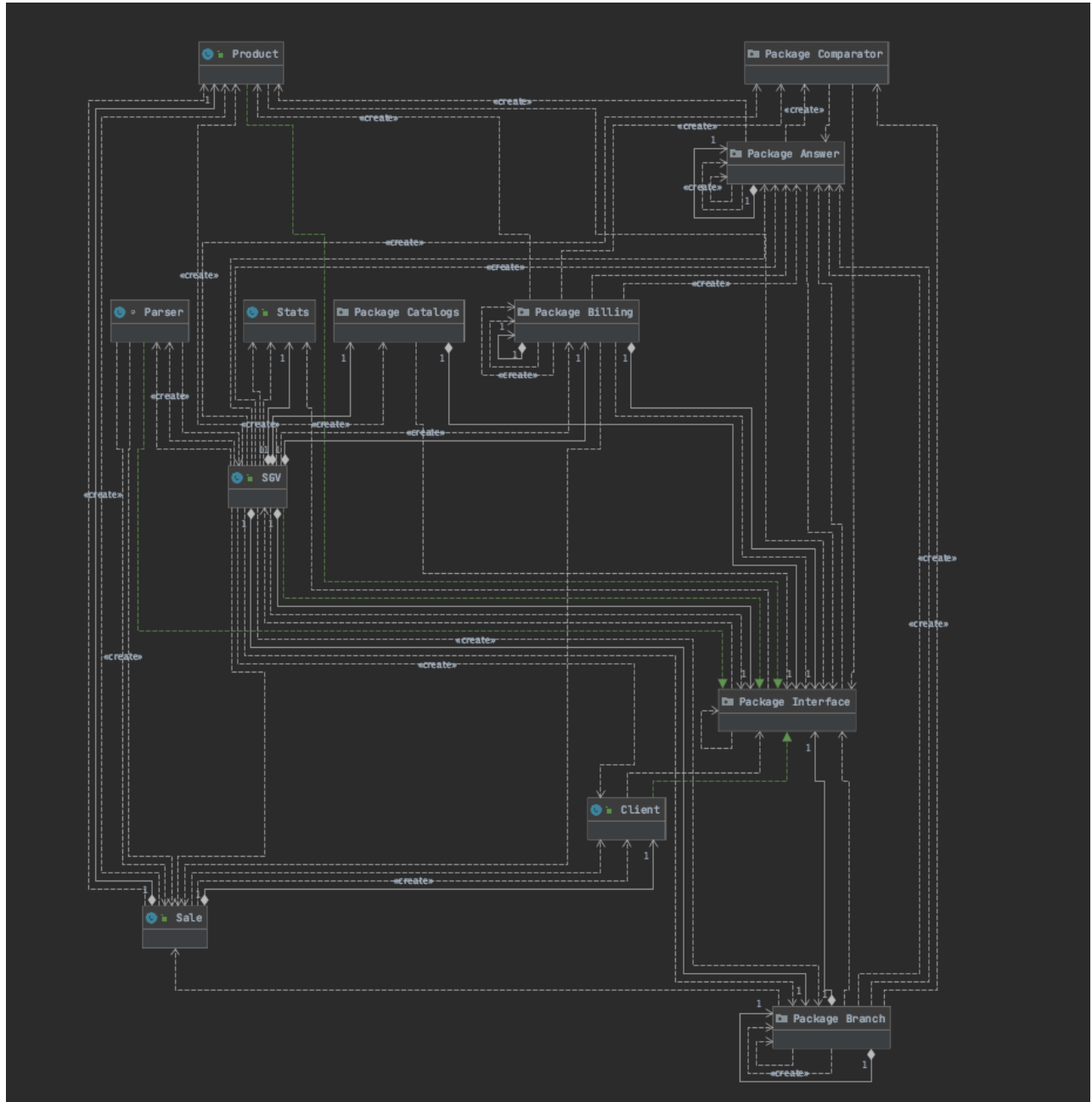


Figura 10: Diagrama de classes completo