Laboratórios de Informática III
TP - Java

***

Docentes: Fernando Martins

Vitor Fonte

Desenvolvedores:

Daniel Cerveira Furtado Malhadas

A72293

Joel Tomás Morais

A70841

Rui Pedro Mesquita Miranda

A75488

Tabela de Conteúdos

1.Introdução

***

1.1.Contextualização

O presente relatório serve de apoio à compreensão do projeto de Laboratórios de Informática realizado no presente ano letivo 2015/2016.

1.2.Motivação e objetivos

Com este projeto pretende-se desenvolver um sistema capaz de emular o funcionamento de um supermercado devendo ser capaz de:

- Armazenar todos os produtos;
- Armazenar todos os clientes;
- Reconhecer 3 filiais do supermercado em questão;
- Reconhecer dois tipos de regimes de vendas, N de normal e P de promoção;
- Armazenar quantidade global ou por filial de compras em que participa cada produto e cada cliente participam, quantidade vendida ou comprada e faturação;
- Registar relação de compra entre produto e cliente assim como entre cliente e produto.

Para organizar estes dados criaram-se 3 classes:

- Catálogo;
- Faturação;
- Filial.

Tendo em conta esta organização dos dados (sendo que cada classe será melhor explicada no próximo capítulo) o nosso sistema terá que suportar as seguintes funcionalidades, a que chamaremos de queries:

- Obter os dados referentes ao último ficheiro de vendas lindo;
- Obter de obter dados gerais respeitantes aos dados atuais registados nas estruturas;
- Obter a lista global dos produtos que nunca foram comprados;
- Obter, para um determinado mês, o número total global de vendas realizadas e o número de clientes compradores.
- Obter, para um determinado cliente, dados mensais sobre o mesmo;
- Obter, para um determinado produto, dados mensais sobre o mesmo;
- Obter todos os produtos comprados por determinado cliente, globalmente ou numa determinada filial, ordenados por quantidade;
- Obter a lista dos X produtos mais vendidos, global ou por filial;
- Obter o top X compradores, global ou por filial;
- Obter a lista dos X clientes que compraram mais produtos distintos;
- Obter a lista dos X clientes que mais compraram determinado produto;

1.3.Estrutura do relatório

No decorrer deste relatório tivemos em consideração duas questões que entendemos ser fundamentais e que explicamos de seguida:

- Estruturação:

Entendemos que não faz sentido começar imediatamente a construir "qualquer coisa" ou rápidamente começar a pensar nas mais complicadas maneiras de chegar ao fim se isso apenas significa ter algo mal pensado e mal estruturado que terá que vir a ser mudado no futuro. Não interessa quão bons sejam os programadores ou quão complexo seja o modelo do projeto que terão criado, se não tiver sido bem estruturado de raiz então apenas irá dar azo a uma incessante necessidade de modificações que poderão vir a pôr em risco a integridade do que já estava feito, criando necessidade de refazer partes significativas do projeto a longo prazo com a falsa esperança de "remediar" problemas que poderiam ter sido facilmente evitados numa fase de análise inicial. Isso é um exemplo de má estruturação e análise dos requisitos que sabemos ter de evitar a qualquer custo, visto que qualquer

tempo "perdido" nesta fase será um indubitável ganho exponêncial em tempo na implementação. Por esta razão temos a noção da enorme importância que este primeiro obstáculo tem e tivemos todo o cuidado para o levar a sério e para o não apressar e isso demonstramos nas estruturas que desenvolvemos já explicitadas no próximo capítulo.

- Simplicidade:

É de grande importância para nós que a mensagem que desejamos transimitir com este relatório seja percebida na sua integridade sem que haja possibilidade de interpretações erradas ou alternativas. Sabendo nós que grande parte dos problemas com o software hoje em dia baseia-se na má comunicação entre os desenvolvedores e os seus empregadores quisemos que todos os esquemas e explicações neste relatório fossem sucintos, diretos ao assunto e simples de perceber. Desta forma emulamos a estrutura de um relatório que poderia ser relativo a um trabalho no mundo de tabalho fora da Universidade e onde o nosso empregador, mesmo com poucos conhecimentos na área seria, mesmo assim, capaz de ler, perceber e fazer uma crítica constructiva sobre a direcção do projeto. Isto permite evitar interpretações erradas do objetivo por parte dos programadores e também permite ao empregador compreender melhor se realmente os seus objectivos podem ser todos realizados da forma que idealizou ou se são demasiado optimistas.

2.Principais Classes

***

2.1.Catálogo

Classe que, a partir dos ficheiros lidos, **contém todos os produtos ou clientes importados**. Para a escolha da estrutura tinhamos a possibilidade de escolher um dos seguintes tipos:

- Map
- List
- Set

Como os produtos ou os clientes se **encontram identificados somente por um código alphanumérico** interpretado como um objeto do tipo **String**, entendemos que **não devem ser tolerados nem guardados códigos repetidos**, pois tal não teria significado no contexto da aplicação que se pretende desenvolver. Como tal, uma estrutura do tipo **List fica fora de questão**, pois se quisessemos evitar repetidos teriamos sempre de iterar pela lista para confirmar uma inserção, ou entao usá-la como argumento de um construtor de um set que seria o argumento do construtor de uma nova lista etc... Teriamos trabalho completamente desnecessário e mais do que redundante.

Da mesma forma, pouco ou até **nenhum sentido haveria em utilizar uma estrutura Map**. Já que apenas identificamos as entidades cliente ou produto pelos seus códigos alphanuméricos iriamos mapear esses códigos com o quê? A única coisa a fazer que tivesse algum sentido no contexto desta aplicação usando um Map seria mapear os códigos alphanuméricos com eles próprios, mas no fundo... Porque razão fariamos isso?

Resta então somente mais uma hipótese, a **utilização de uma estrutura de tipo Set** que, no fundo, é indubitávelmente a mais indicada. Um Set **não permite repetições** exatamente como

queriamos e a sua utilização **encaixa perfeitamente no contexto do problema**. As desvantagens do Set que acabam principalmente por ser impossibilidade de obter um determinado objeto inserido sem iterar (poderiamos querer o endereço de um inserido para mudar alguma coisa) são algo que não têm o mínimo impacto no projeto e portanto aceitamos sem medo que qualquer prejuízo nos assombre mais tarde por esta decisão.

No entanto as escolhas não terminam por aqui, falta algo também bastante relevante, que tipo de Set é o mais indicado?

Temos:

- HashSet
- TreeSet
- LinkedHashSet

Antes da escolha devemos entender para o que a nossa estrutura servirá, depois de ler as queries a implementar entendemos que **maioritáriamente a estrutura será inquirida sobre a existência de um determinado código ou então um código será inserido**.

Sabemos que um **TreeSet** é o mais indicado para quando queremos ordenar os valores que inserimos, daí ser uma estrutura caracterizada pelo nome de Tree. No entanto, como vemos no paragrafo anterior não temos nenhum propósito para esta funcionalidade, ainda por cima quando nos reduz a eficiência em relação a outros tipos de Set's (exatamente por ter a necessidade de ordenar ao inserir) para algo de **complexidade do género de O(log(n)) na inserção e na procura**. Afirmamos então que **TreeSet não é o que procuramos**.

Temos depois um **LinkedHashSet** que acaba por ser uma lista duplamente ligada entre cada elemento permitindo que os elementos possam ser iterados por uma ordem específica, mais uma funcionalidade que não teria propósito nesta aplicação, portanto **mais uma estrutura que não é de todo a mais indicada**.

Por fim temos o **HashSet** que nos **permite consultar elementos e inserir com a fantástica complexidade de O(1)** algo que rapidamente entendemos mal reparamos no nome Hash atribuído a este Set. Com um HashSet **podemos ainda indicar um tamanho inicial para o seu tamanho** (assim como um load factor), isto possibilita que, se soubermos a quantidade de elementos a inserir (o que sabemos, basta contar o número de linhas lidas do ficheiro txt correspondente) podemos tornar esta estrutura o mais estática possível, não havendo a necessidade de alocar nova memória sempre que se quisesse inserir novo elemento já que esta é alocada de imediato internamente pelo construtor do objeto HashSet ao se fornecer uma capacidade inicial com um load factor igual a 1 (não queremos re-hashes já que não iremos inserir nem mais nem menos que a capacidade inicial indicada), estas características fazem do **HashSet a estrutura mais indicada para guardar os códigos do catálogo**.

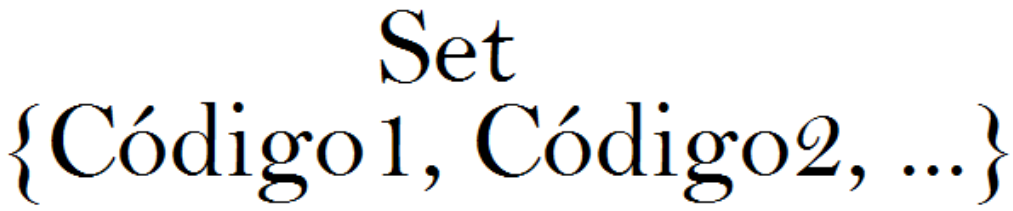

Figura 1 - Representação da Estrutura da classe Catalogo

2.2.Faturação

Módulo de dados que irá conter as estruturas de dados responsáveis pela resposta eficiente a questões quantitativas que **relacionam os produtos às suas vendas mensais, em modo Normal (N) ou em Promoção (P)**, para cada um dos casos **guardando o número de vendas e o valor total de facturação** de cada um destes tipos. Este módulo deve referenciar todos os produtos, mesmo os que nunca foram vendidos. **Estima-se que os produtos e os seus dados serão consultados/modificados inúmeras vezes**. Tinhamos a possibilidade de escolher entre:

- Set
- List
- Map

O Set não será o mais indicado visto que não seria possível criar uma relação entre o código de um produto e os seus dados sem criar uma classe que contenha esse código e os dados, no entanto como fariamos para encontrar os dados de um determinado produto? Pois teriamos de iterar sobre o Set algo que poderia ter uma complexidade tão abomínável quanto O(n).

A List também não é indicada essencialmente pelo mesmo exato propósito referido para o Set, para além de permitir repetições.

Usamos então um Map, Este irá permitir que se mapeie cada código de produto com os seus dados respetivos, evitando a repetição de códigos de produto e contendo o método bastante útil get() que irá permitier obter os dados de determinado produto.

Temos então que escolher que tipo de Map iremos usar:

- TreeMap
- LinkedHashMap
- HashMap

Não pretendemos nenhuma ordem específica das entradas do nosso Map, isto poderia até ser interessante em algumas queries, no entanto essa ordem pode variar, novas queries poderiam até ser adicionadas no futuro que não fossem tão rápidas usando essa ordenação inicial e a inserção seria mais lenta com **uma complexidade do género de O(log(n)) ao inserir ou consultar**. Logo um **TreeMap não é o aconselhável**.

Temos depois um **LinkedHashMap** que acaba por ser uma lista duplamente ligada entre cada entrada permitindo que os elementos possam ser iterados por uma ordem específica, mais uma funcionalidade que não teria propósito nesta aplicação, portanto **mais uma estrutura que não é de todo a mais indicada**.

Sobra somente o HashMap que nos **permite consultar o valor de determinda chave e inserir com a fantástica complexidade de O(1)** algo que rapidamente entendemos mal reparamos no nome Hash atribuído a este Map. Com um HashMap **podemos ainda indicar um tamanho inicial para o seu tamanho** (assim como um load factor), isto possibilita que, se soubermos a quantidade de elementos a

inserir (o que sabemos, basta contar o número de linhas lidas do ficheiro txt correspondente) podemos tornar esta estrutura o mais estática possível, não havendo a necessidade de alocar nova memória sempre que se quisesse inserir novo elemento já que esta é alocada de imediato internamente pelo construtor do objeto HashMap ao se fornecer uma capacidade inicial com um load factor igual a 1 (não queremos re-hashes já que não iremos inserir nem mais nem menos que a capacidade inicial indicada), estas características fazem do **HashMap a estrutura mais indicada para guardar a relação entre os códigos de produto e os seus dados**.

Falta-nos agora então entender o que são estes “dados” de produtos que tanto falamos.

Estes dados são representados por um conjunto de objetos instânciados apartir da classe **DadosFilial (dados relativos a uma filial)**. Dizemos um conjunto, pois esta aplicação deve suportar várias filiais e, embora no enunciado fale apenas em três, nunca sabemos o futuro, portanto achamos boa prática e não conseguiriamos de todo sequer convencer-nos a nós próprios a programar esse pormenor das filiais de uma maneira que não fosse **escalável ao ponto de podermos ter dados não só relativos a uma filial como também a 0, 1, 2, 3, 4, 5, 6, 7, 8, ...** Isto trás outra vantagem. Sabemos que iremos guardar quantidades absurdas de informações na ordem dos milhares de mapeamentos chave/valor, por isso, imagine-se um produto nunca comprado, se podermos evitar instanciar estas classes de dados na filial com valores a zero que nunca terão propósito nenhum isso será mais do que benéfico em dois campos diferentes:

- Menor tempo de Inserção.
- Menor RAM necessária atribuir á aplicação.

Note-se também que **este conjunto de dados de filiais é representado por um Map**. Neste Map as chaves são o número identificador da filial e os valores os dados da filial correspondentes á chave indicada. Mais uma vez usamos um mapeamento por várias razões, a primeira é que **não queremos repetições**, por essa razão uma **List fica logo fora de questão** e queremos ser capazes de rapidamente **obter os dados de uma filial específica sem ter de iterar** para saber qual é qual, coisa que um **Set não é capaz de fazer**.

Por sua vez, estes objetos **DadosFilial** irão, pela mesma razão descrita para o mapeamento das filiais, conter mapeamentos entre cada regime de compra e um mapeamento entre o mês e um objeto que abstrai **a quantidade comprada, número de vendas distintas e faturação gerada** do produto em questão. Este objeto será uma instância da classe **TriploQuantidadeVendasFaturacao**

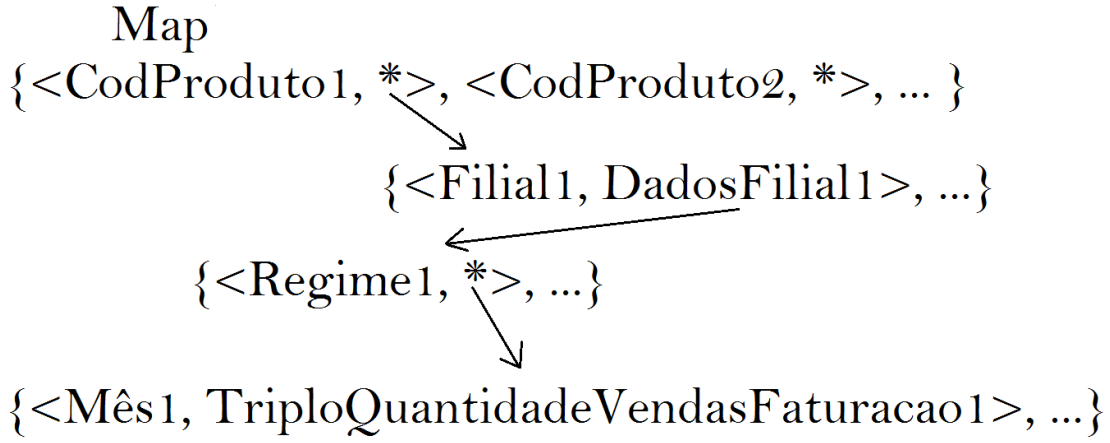

Figura 2 - Representação da Estrutura da Classe Faturação

2.3.Filial

Módulo de dados que, a partir dos ficheiros lidos, conterá as estruturas de dados adequadas à representação dos relacionamentos, fundamentais para a aplicação, entre **produtos e clientes**, ou seja, **para cada produto, saber quais os clientes que o compraram, quantas unidades cada um comprou, em que mês e em que filial**. Para a estruturação optimizada dos dados deste módulo de dados será crucial analisar as queries que a aplicação deverá implementar, tendo sempre em atenção que pretendemos

ter o histórico de vendas organizado por filiais para uma melhor análise, não esquecendo que **existem 3 filiais nesta cadeia**.

Este pormenor das várias filiais é bastante relevante sendo de interesse referir que o nosso projeto, como referido no sub-capítulo anterior deve suportar não só as três filiais referidas no guião do projecto, mas deve também ser escalável ao ponto de suportar um número variável das mesmas. Seja esse número zero ou até mesmo na ordem dos milhões. Por essa razão temos então cada filial guardada numa estrutura ArrayList capaz de crescer para conter um maior número de filiais.

Antes da escolha da estrutura a usar em cada Filial, temos de realmente entender o que queremos estruturar. O mínimo essencial que necessitariamos seria uma estrutura que contivesse a relação entre cada cliente e produtos comprados pelo mesmo. Esta acaba por ser a parte mais importante da estrutura na classe Filial já que a maioria das queries que a consultam acedem através de um código de cliente. Isto seria pois suficiente para responder a todas as queries. A inserção seria rápida, mas seriam as queries rápidas o suficiente? Depois de testar, sabemos que não, certas queries, para usufruirem de um bom tempo de execução exigem que se aceda com um código de produto. Nada nos impede então, a não ser possíveis problemas com os tempos de inserção (melhor explicado no sub-capítulo Vendas), de adicionar outra parte da estrutura como sendo um relacionamento entre cada produto comprado e os clientes que o compraram.

Já temos uma estrutura melhor pensada, mas, para certas queries não é ainda suficiente. Isto porque, se queremos fazer assim teremos de sobrecarregar esses relacionamentos referidos com contadores para quantidades, vendas, faturação, etc... que irão tornar a inserção bastante mais lenta e o acesso e possível ordenação também. Por essa razão, certos contadores teve-se a ideia de guardar num relacionamento à parte, estamos a falar do relacionamento entre cada mês, clientes distintos que compraram nesse mês, produtos distintos que comprou (esse cliente) e dados sobre cada compra.

Para a decisão das estruturas a usar para estes relacionamentos tinhamos as seguintes opções:

- Set
- List
- Map

O Set não será o mais indicado visto que não seria possível criar uma relação entre os códigos ou o mês e os seus dados sem criar uma classe que contenha esse código/mês e os dados, no entanto como fariamos para encontrar os dados de um determinado produto, por exemplo? Pois teriamos de iterar sobre o Set algo que poderia ter uma complexidade tão abomínável quanto O(n).

A List também não é indicada essencialmente pelo mesmo exato propósito referido para o Set, para além de permitir repetições.

Usamos então um Map, Este irá permitir que se mapeie cada código ou cada mês com os seus dados respetivos, evitando a repetição de códigos de produto e contendo o método bastante útil get() que irá permitier obter os dados de determinado produto.

Temos então que escolher que tipo de Map iremos usar:

- TreeMap
- LinkedHashMap
- HashMap

Não pretendemos nenhuma ordem específica das entradas do nosso Map, isto poderia até ser interessante em algumas queries, no entanto essa ordem pode variar, novas queries poderiam até ser adicionadas no futuro que não fossem tão rápidas usando essa ordenação inicial e a inserção seria mais lenta com **uma complexidade do género de O(log(n)) ao inserir ou consultar**. Logo um **TreeMap não é o aconselhável**.

Temos depois um **LinkedHashMap** que acaba por ser uma lista duplamente ligada entre cada entrada permitindo que os elementos possam ser iterados por uma ordem específica, mais uma

funcionalidade que não teria propósito nesta aplicação, portanto **mais uma estrutura que não é de todo a mais indicada**.

Sobra somente o **HashMap** que nos **permite consultar o valor de determinda chave e inserir com a fantástica complexidade de O(1)** algo que rapidamente entendemos mal reparamos no nome Hash atribuído a este Map. Com um HashMap **podemos ainda indicar um tamanho inicial para o seu tamanho** (assim como um load factor), isto possibilita que, se soubermos a quantidade de elementos a inserir podemos tornar esta estrutura o mais estática possível, não havendo a necessidade de alocar nova memória sempre que se quisesse inserir novo elemento já que esta é alocada de imediato internamente pelo construtor do objeto HashMap ao se fornecer uma capacidade inicial com um load factor igual a 1 estas características fazem do **HashMap a estrutura mais indicada**.

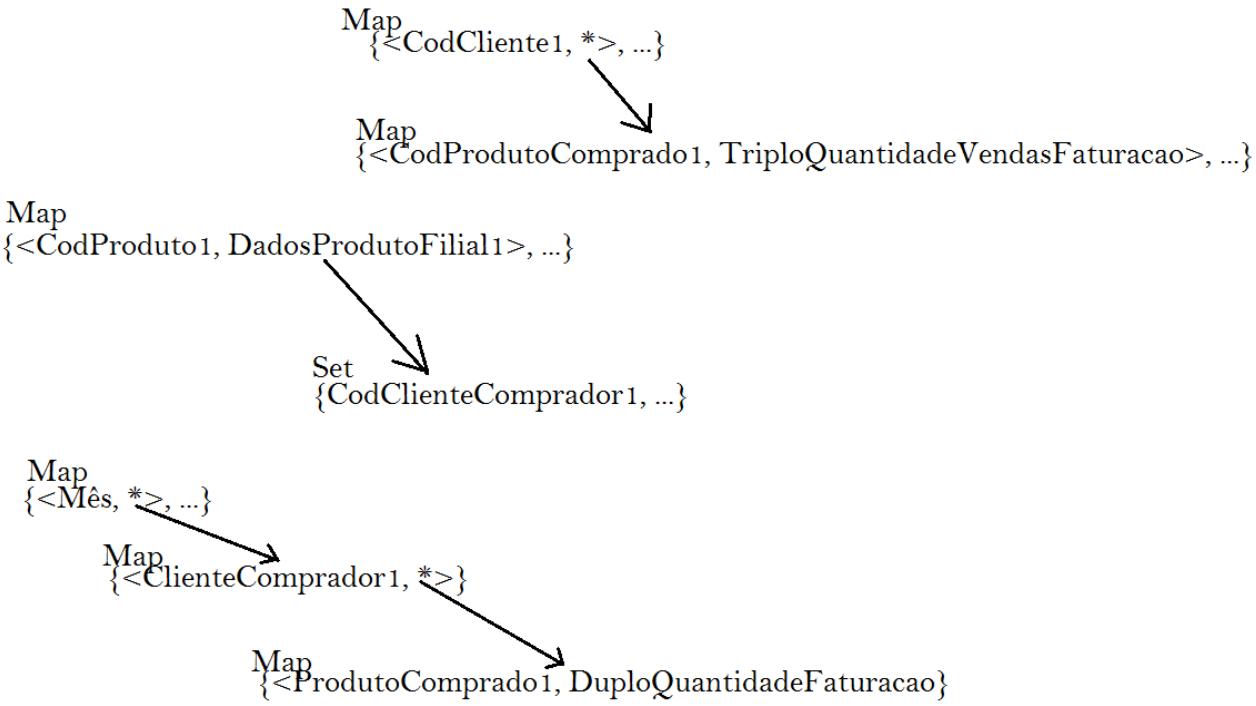

Figura 3 - Representação das Estruturas na Classe Filial

2.4.Vendas

A classe vendas existe com o propósito de validar as linhas já préviamente lidas do ficheiro de vendas e assim popular a classe Venda (que irá conter os dados de uma linha já validada) que é por sua vez inserida na Faturação e na Filial respetiva.

Os dados validados a inserir na classe Venda são:

- Cliente;

- Produto;
- Quantidade;
- Mês;
- Preço;
- Filial;
- Regime;

Na validação tem-se em conta os seguintes critérios:

- O cliente tem que existir no catálogo de clientes(que já foi préviamente instânciado e populado). Como funcionalidade opcional o número de caracteres e a sua posição também é avaliada para determinar se realmente é um código correto;
- O Produto deve existir no catálogo de produtos (que já foi préviamente instânciado e populado). Como funcionalidade opcional o número de caracteres e a sua posição também é avaliada para determinar se realmente é um código correto;
- O mês deve estar entre os valores indicados na classe ValoresFixos (por defeito são 0 e 12);
- A filial deve estar entre os valores indicados na classe ValoresFixos (por defeito são 1 e 3);
- A quantidade deve estar entre os valores indicados na classe ValoresFixos (por defeito são 0 e 200);
- Os regimes devem ser um dos indicados na classe ValoresFixos (por defeito existe regime N e regime P);
- O preço deve estar entre os valores indicados na classe ValoresFixos (por defeito são 0 e 999.99).

Se algum destes critérios de validação falhar então é lançada um exceção para, de forma elegante se detetar essa falha. Temos então exceções diferentes para cada tipo de erro o que nos permite uma maior facilidade na atualização de contadores relevantes, como o número de produtos inválidos, linhas inválidas, etc...

É de interesse referir que, para a validação e inserção é usado código multithread através da abstração fornecida pelo método parallelStream(). Este método permite que cada core do computador lance uma thread e que a inserção seja realizada concurrentemente. Por um lado isto é sem dúvida fantástico. Reduz o tempo de inserção necessário de uma maneira absurda. No entanto é bastante perigoso, pois é necessário implementar os mecanismos necessários para evitar situações como o erro clássico que iremos descrever:

Uma thread pretende adicionar somar o valor 1 a uma variável que tem o valor zero. A meio deste processo (já leu que o valor presente é zero, mas ainda não adicionou o um) a execução desta thread é temporáriamente bloqueada e outra thread começa a executar o mesmo bloco de código. Agora esta thread quer somar 1 também. Ao ler o valor da variável nota que é 0 (a outra thread ainda não o mudou) por isso adiciona 1 e o valor passa a ser 1. De repente a thread anterior acorda e continua o que estava a fazer, já tinha lido que o valor da variável era 0 portanto adiciona 1 e o valor passa a ser 1... Perdeu-se então o que a primeira thread executou.

A solução que usamos na validaçao para lidar com este problema foi recorrer á primitiva de Java, **synchronized.** Assim podemos declarar métodos para mudar o valor de cada contador necessário com esta primitiva no seu protótipo tendo que agora já nunca nenhuma thread poderá intervir dessa forma com a execução de outra que esteja a mudar o valor do mesmo contador.

Os métodos para inserir são também synchronized, mas os tempos de inserção poderiam ser ainda melhores caso se usassem ConcurrentHashMaps nas estruturas em vez de HashMaps e se controlasse o acesso a inteiros com recurso a métodos como na validação.

Neste momento a concurrência, a inserir apenas é aproveitada para inserir na Faturação e na Filial simultâneamente.

2.5.Diagrama de Classes

Segue-se o Diagrama de Classes do nosso projeto sendo que, como o objetivo acaba por ser ajudar a perceber o projeto, o diagrama foi simplificando sendo que classes sem importância para a compreensão da estrutura do mesmo foram retiradas. Por exemplo exceções e wrappers.

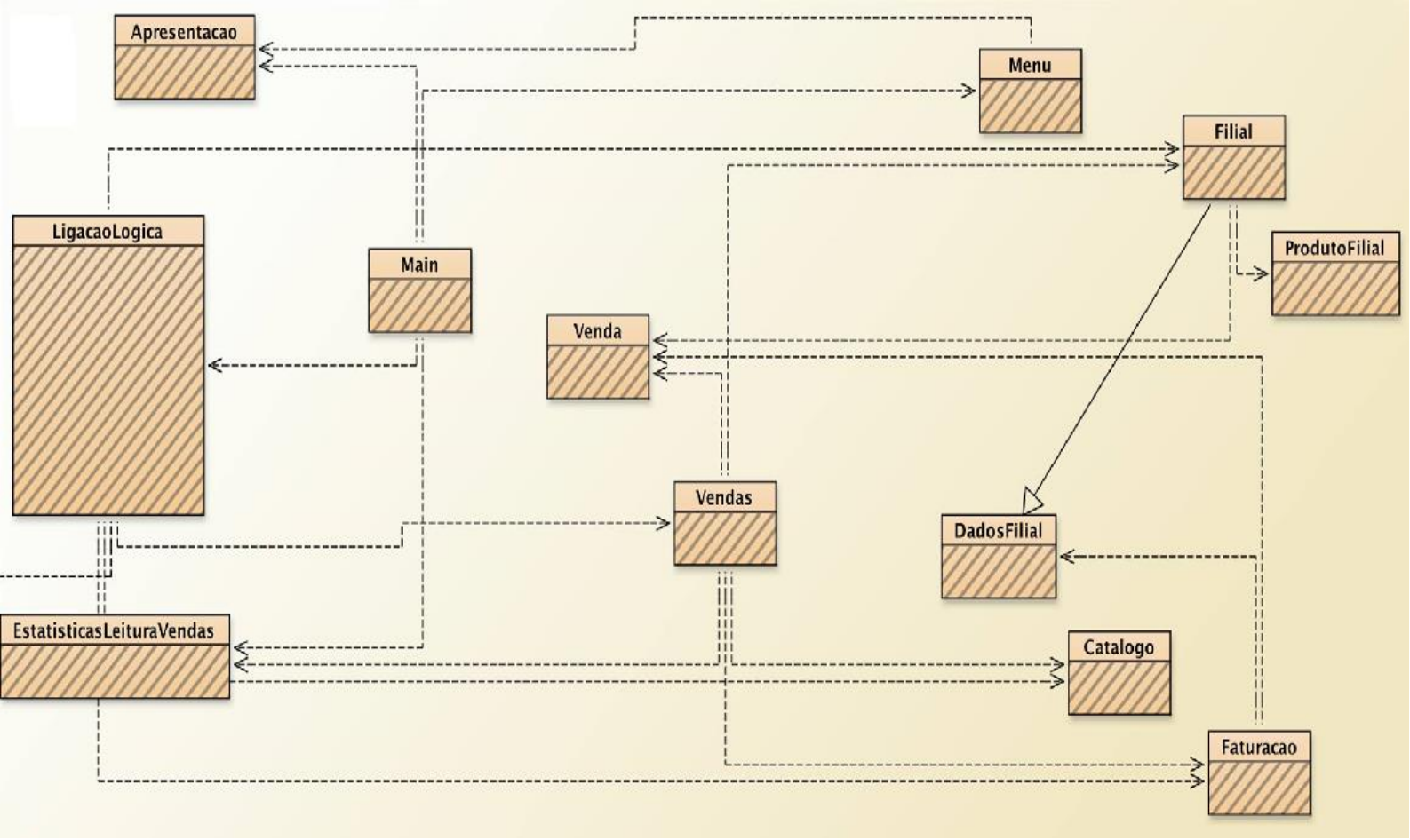

Figura 4 - Diagrama de Classes do Projeto

3.Interface

***

3.1.Utilização

O programa a que se dá o nome de ***GereVendas*** é um **programa não para programadores mas sim para utilizadores**. Por mais banal que esta frase pareça nunca é demais frisar a colossal importância do seu significado. Até porque **um programa não tem propósito de existir se o seu público alvo não o saber usar**. Por esta razão tivemos o consciente esforço de tornar a utilização do nosso programa o mais fácil possível, assim como o menos cansativa possível. São duas características que achamos essencial qualquer projeto deste tipo ter.

Não deve ser cansativo. Exemplos de um programa cansativo seria por exemplo um programa que ao apresentar listas grandes de items não permitisse um rápido acesso a qualquer página ou fácilmente voltar atrás. O nosso programa **permite sempre o cancelamento da visualização de qualquer lista assim como a escolha da página para onde se pretende saltar**. Outro exemplo é um programa que seja demasiado 'banal' visualmente, por exemplo, um programa só com letras, menus mal formatados e opções para nomes de comandos mal pensados. **Os nossos menus são todos envoltos em caixas, sendo cada campo a demonstrar devidamente separado de informação distinta** de modo a possibilitar uma mais fácil compreensão. **Os nomes dos comandos foram também estudados de modo a que fossem intuitivos**, por exemplo, em qualquer momento pode escrever-se 'h' para obter a lista dos comandos possíveis no menu presente. Sendo 'h' a primeira letra de 'help' este comando torna-se bastante intuitivo.

Deve ser fácil. Na nossa interface o **utilizador nunca se sentirá perdido** sendo que em momento algum olha para o ecrã sem que lhe seja indicada pelo menos uma tecla a carregar para prosseguir e **poderá sempre cancelar a acção que está a realizar nesse momento** assim como em momento algum tem a necessidade de comprênder minimamente o funcionamento interior do programa (a sua programação), com isto queremos afirmar, **qualquer pessoa seria capaz de usar o nosso programa**, até mesmo alguém que nunca tenha sequer ouvido falar em programação.

De notar que o nosso programa lê de imediato os ficheiros após ser iniciado, sendo questionado o utilizador sobre os nomes dos mesmos, de seguida qualquer uma das queries pode ser executada sem nenhuma restrição, sendo que **inputs errados para as queries são sempre detetados como tal** e a interface volta a solicitar o input ao utilizador em vez de o programa terminar inesperadamente.

Sabemos também que muitas vezes interfaces podem se tornar demasiado complexas de ler a nível de programação, algo que não acontece no nosso projeto sendo que tivemos o cuidado de usar "switches" em vez do uso abusivo e menos legível de "if's" assim como nas opções que tomamos de aumentar o número de linhas usadas em certas partes para evitar linhas de tamanhos absurdos e exagerados.

A programação que gera os Menus e tudo o que é apresentado no ecrã ao utilizador está também separado de tudo o resto numa classe específica só para si para facilitar coisas como a dobragem para outras linguagens (inglês, francês, etcc...) sem que o tradutor tenha de perceber minimamente de programação

As funcionalidades/queries encontram-se já explicitadas na introdução, no sub-capítulo **Motivação e Objetivos**, caso se queira saber sobre as mesmas.

4.Testes deDesempenho e análise dos Resultados

***

Importa referir que **todos os tempos que se apresentam neste relatório estão em segundos e não são o resultado de uma só medição mas sim da média obtida ao fim de 1000 medições**.

4.1.Leitura dos ficheiros

Antes de lermos os ficheiros de texto surgiu a dúvida de como os iriamos ler. Na liguagem de programação Java temos acesso a duas excelentes abstrações para o fazer:

- BufferedReader
- Scanner

Tivemos então de testar a leitura dos ficheiros para cada um deles e ver qual seria o mais indicado. Apresentamos os resultados nas tabelas que se seguem:

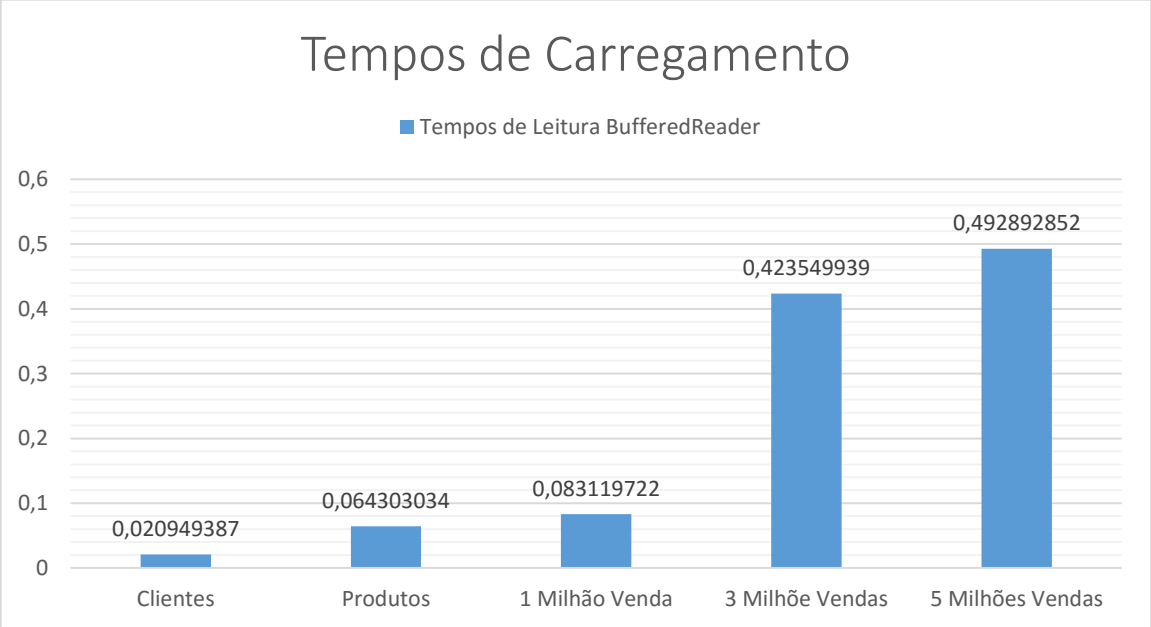

Tabela 1 – Tempos de Leitura dos Ficheiros com BufferedReader

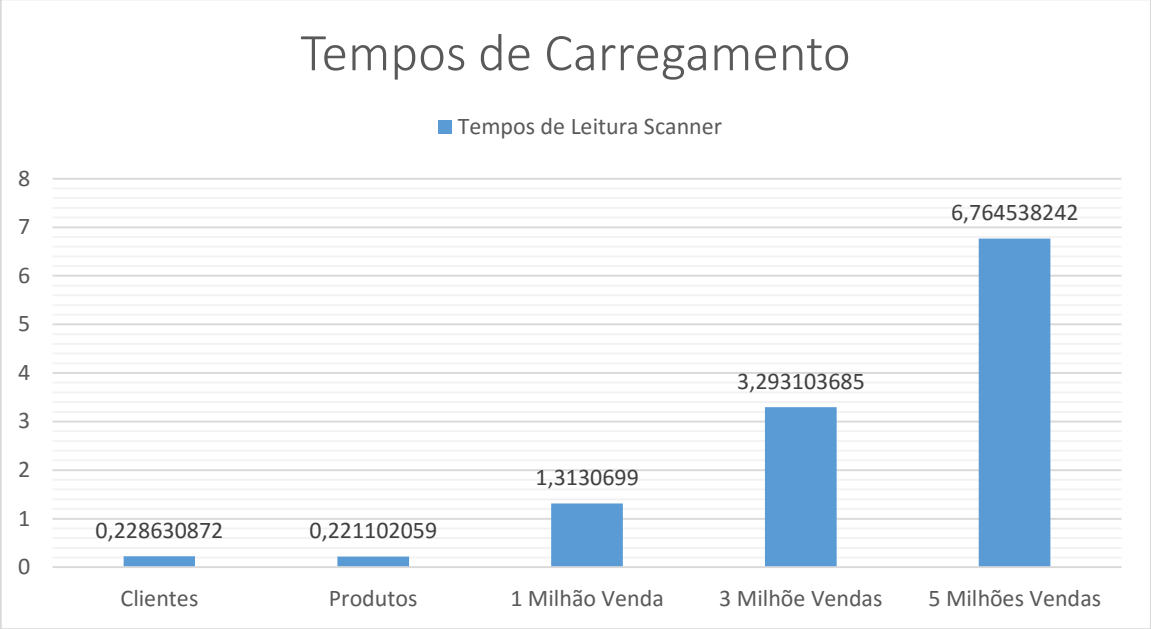

Tabela 2 – Tempos de Leitura dos Ficheiros com Scanner

Podemos então notar que a leitura com recurso a um BufferedReader é bastante mais eficiente, por essa razão foi esse o método usado para a leitura.

Apresentamos de seguida um gráfico de barras com os tempos de leitura e incerção de cada ficheiro.

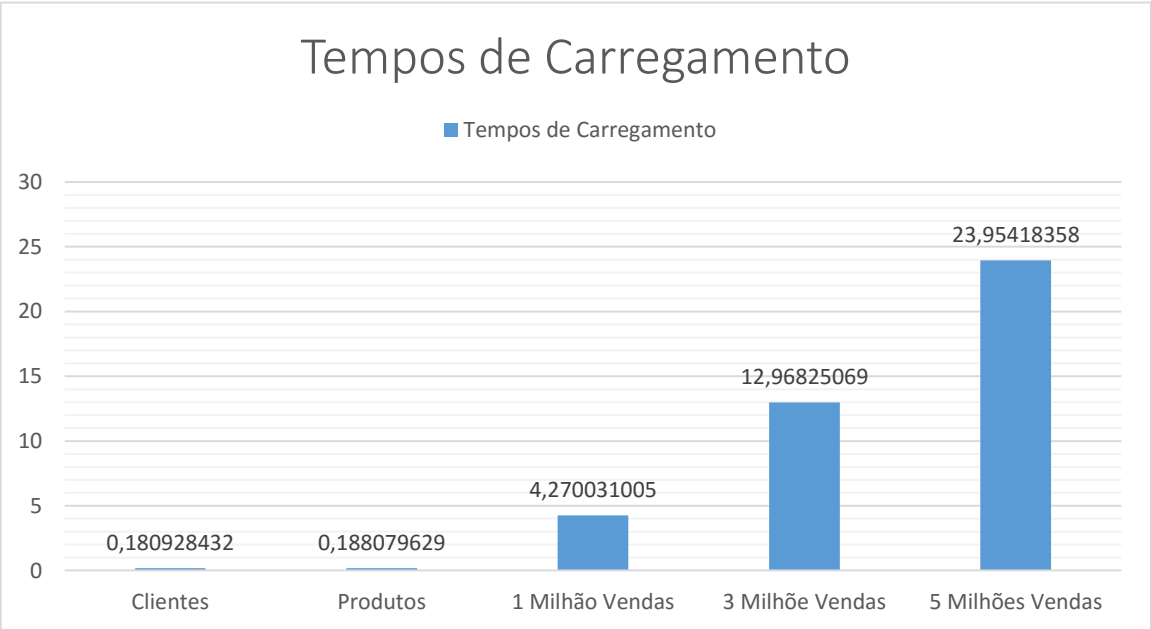

Tabela 3 – Tempos de Carregamento TP-Java

Vemos que **a leitura "menos eficiente" acontece ao ler os ficheiros de vendas**. No entanto tal justifica-se ao estudar a estrutura filial explicada no capítulo anterior, vemos que a complexidade da estrutura justificaria até um tempo mais elevado de inserção, sendo que estes tempos apenas se

conseguem obter graças à estratégia de lançar threads com recurso a parallelStream() aquando da inserção como indicado no sub-capítulo sobre a classe Vendas do capítulo anterior.

Comparando com os tempos do trabalho de C:

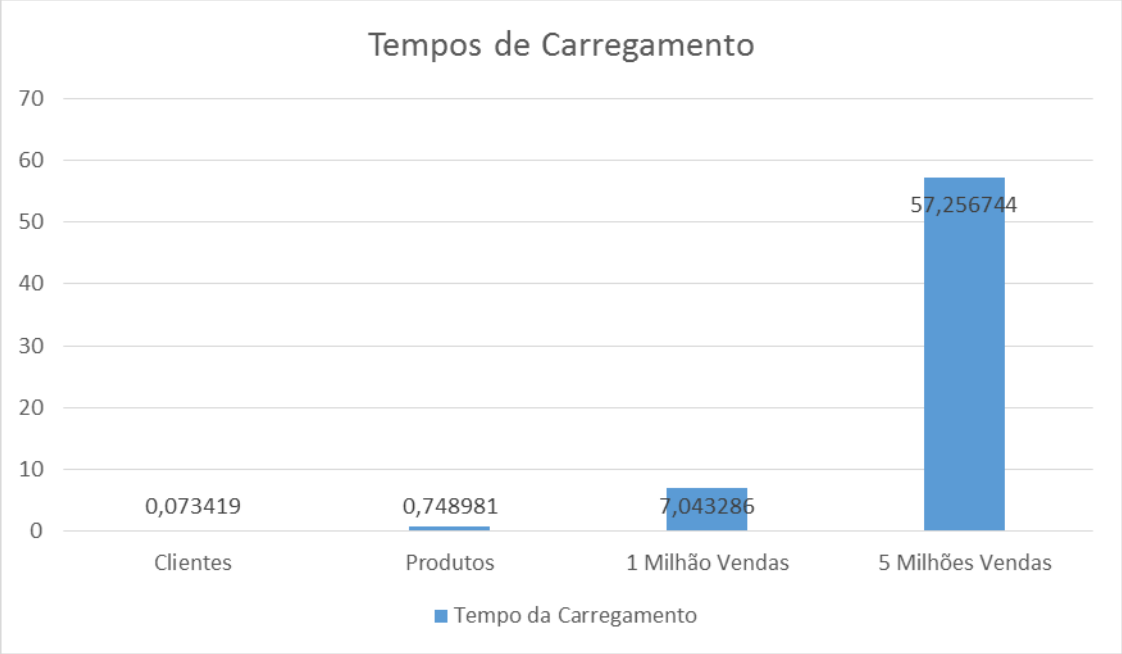

Tabela 4 – Tempos de Carregamento TP-C

Podemos notar que a inserção é bastante mais eficiente neste trabalho de Java do que no de C, tal acontece maioritáriamente pela utilização de métodos introduzidos no atual Java 8 que melhoram de uma forma estrondosa a utilização de Collections de Java. Inicialmente tinhamos pensado que este não seria o resultado final, pois entendemos que a linguagem C é de muito mais baixo nível do que Java possibilitando-nos um muito maior controlo sobre o que realmente acontece. Por isso fomos apanhados de surpresa ao notar os tempos que obtivemos, mas atribuímos esse sucesso estrondoso, como já referimos, aos novos métodos que abstraem a utilização de, por exemplo, multithreading e que se pretendessemos implementar em C talvez o nosso código deixasse de ser “standard”.

4.2.Mapa e Desempenho de Queries

Antes de apresentar os tempos de cada query em cada ficheiro de vendas apresentamos um mapa que mostra os módulos utilizados por cada uma, assim como o input que necessita e o output que se espera

Query	Produtos	Clientes	Faturação	Filial	Resultado
1	Consulta de valor	Consulta de valor	Consulta de valor	Consulta de valor	Dados do último ficheiro de vendas lido
2				Consulta de valor	Dados globais mensais
3			Varrimento		Lista de produtos não vendidos
4				Mês	Vendas realizadas+Compradores distintos num mês
5				Cliente	Dados mensais
6				Produto	Dados mensais
7				Consulta de todos os produtos comprados por cliente + filial + Cliente	Lista de produtos mais comprados por cliente
8				Consulta de todos os produtos + filial + X	Lista ordenada por quantidade dos X produtos mais comprados
9				Consulta de todos os clientes + X + filial	Lista ordenada dos X clientes mais compradores
10				Consulta de todos os clientes + X + filial	Lista ordenada dos X clientes que compraram mais produtos distintos
11				Consulta de todos os clientes que compraram produto + produto + filial + X	Lista ordenada dos X clientes que mais compraram o produto

Tabela 5 – Mapa de queries

Com este mapa em mente, olhamos agora para os tempos de cada query

Query	Vendas_1M	Vendas_3M	Vendas_5M	Input do utilizador
Q1	0.074658199	0.054926765	0.058309064	
Q2	0.011248958	0.011468343	0.011710394	
Q3	0.039259734	0.042401693	0.043322	
Q4	0.014424274	0.01471636	0.009012209	Mês 9
Q5	0.006455406	0.007389398	0.239269143	Cliente Z5000
Q6	0.141912032	0.17813501	0.14238587	Produto AF1184
Q7	0.071237838	0.069893728	0.067803793	Cliente Z5000 + todas as filiais
Q8	0.439627087	0.924895759	3.270748178	Todas as filiais + Top 170080
Q9	0.177786473	0.334498685	0.493165266	Todas as filiais + Top 16384
Q10	0.090263648	0.071307974	0.096491543	Todas as filiais + Top 16384
Q11	0.031150598	0.036846494	0.042904184	Todas as filiais + Top 16384 + Produto AF1184

Tabela 6 – Tempos de Execução das Queries

De notar o cuidado tido para que, com o aumento exponêncial do tamanho do ficheiro das vendas, os tempos das queries não aumentassem em demasia. **É aqui que justificamos a complexidade aparentemente exagerada do módulo filial**, podemos observar que as únicas queries que realmente aumentam um pouco o tempo de espera por parte do utilizador á medida que se aumenta o tamanho do ficheiro vendas são a **8, a 9 e a 5 e 6 também têm possibilidade de crescer até certo ponto**. Sendo que todas **as outras ou atravessam uma coleção somente do tamanho do número de produtos ou do número de clientes ou então apenas consultam valores**, e isto é algo que permanece constante com o mudar de ficheiro de vendas. Note-se também que há um limite para quão grande o tempo das queries referidas 8, 9, 5 e 6 pode crescer, pois foi tido o cuidado na sua implementação para que no máximo fosse preciso uma travessia de uma coleção do tamanho de todos os clientes ou de todos os produtos. Resumindo, **nenhuma query, por maior que seja o ficheiro de vendas, irá fazer uma travessia de uma coleção maior que o número de produtos ou o número de clientes.**

Em grande parte das queries a estrutura utilizada recaiu pelo uso de um ConcurrentHashMap para poder aproveitar a utilização de parallelStream, portanto os testes das estruturas a usar em cada query acabaram por ser bastante reduzidos, pois pouco espaço havia para o uso de outras collections. Sendo que os testes executados inicialmente acabaram por ser abandonados a favor deste código multi threaded, por esse motivo não achamos relevante colocar aqui os seus tempos.

5.Conclusões e Sugestões

Terminada a segunda fase deste projeto, depois de feita uma minunciosa análise de requisitos e avaliação do problema, chegamos a conclusões (apresentadas ao longo do corpo deste relatório) que nos deixaram satisfeitos. Pensamos ter sido capazes de apresentar explicações detalhadas no sentido em que toda a informação essencial para entender a implementação do nosso sistema e o seu funcionamento se encontra presente sem nenhuma exceção, mas ao mesmo tempo bastante simples na maneira como será possível para qualquer programador, sem conhecimento pévio sobre o projeto ou sobre a programação que lhe deu existência , que pretenda usar os módulos criados para este projeto num outro projeto distinto o consiga fazer com toda a facilidade. Com isto pretendemos afirmar que estamos satisfeitos com as conclusões a que chegamos e com o plano de acção que elaboramos tendo conseguido realizar tudo o que foi proposto de forma simples e eficaz sendo que foram ainda realizados vários testes de performance nunca obtendo resultados não aceitáveis.

Tendo isto em conta temos também a noção que nunca nenhum projeto se pode dar por realmente terminado ou por realmente perfeito, há sempre a possibilidade de melhorar algo e esse facto não deve nunca ser ignorado, por mais experiência que venhamos a ter, existe sempre um limite para quão boas as nossas primeiras impressões poderão ser. Desta forma, estamos preparados para uma eventual necessidade de modificar algo que aqui poderá ter sido afirmado e/ou concluído. Isto é, o nosso projeto foi elaborado com a ideia de criar algo que permita uma fácil expansão tanto de funcionalidades, dados (mais filiais, meses, etc...) como de novas ideias que poderiamos vir a ter ou que poderiamos necessitar de implementar na continuação do projeto caso lhe fossemos dar continuidade (algo que aconteceria num projeto de trabalho real fora da Universidade). Isto é essencial em qualquer projeto, pois sabemos que no mundo real o empregador nem sempre sabe de raiz tudo aquilo que realmente quer implementar por isso torna-se natural que, num momento mais avançado da implementação, liste novas funcionalidades que quer ver presentes. Um projeto mal pensado para expansões e modificações poderia ter de ser refeito de raiz para não dar problemas ou graus baixos de eficiência, no entanto tivemos a ideia presente de tentar criar algo que podesse minimizar todas essas possíveis perdas que poderiam acontecer se este fosse um projeto no mundo real e não um trabalho académico protegido pela redoma que acaba sempre por ser uma universidade. Esses cuidados estão explicitados no capítulo 4, onde mostramos que nunca em situação alguma, seja qual for o ficheiro de vendas, alguma query faz mais do que percorrer uma colecção do tamanho de todos os produtos ou de todos os clientes. Para além disso no capítulo 3 explicamos ainda a estruturação do main, onde se teve todo o cuidado para que fosse conciso e concreto, pois sabemos que muitas vezes interfaces podem se tornar demasiado complexas de ler a nível de programação, algo que não acontece no nosso projeto sendo que tivemos o cuidado de usar switches em vez do uso abusivo de “if’s” assim como nas opções que tomamos de aumentar o número de linhas usadas em certas partes para evitar linhas de tamanhos absurdos e exagerados.

Se tivessemos que melhorar algo, a única coisa que sabemos ser capazes de ainda melhorar mais seria a inserção do ficheiro de Vendas, algo que com mais tempo fariamos sem que tal fosse muito complicado. Os métodos para inserir vendas na faturação e nas filiais são neste momento synchronized, mas os tempos de inserção poderiam ser ainda melhores caso se usassem ConcurrentHashMaps nas estruturas em vez de HashMaps e se controlasse o acesso a contadores com recurso a métodos synchronized como na validação. Neste momento a concurrência, a inserir apenas é aproveitada para inserir na Faturação e na Filial simultâneamente.