Universidade do Minho

Escola de Engenharia

Unidade Curricular:

Laboratórios de Informática III

Ano Lectivo de 2014/2015

Relatório de Desenvolvimento de Projecto

GestHiper - Java

70719 Diogo Constâncio,

70513 Pedro Araújo

Índice

Índice de Figuras

Fotos

FIGURA 1 – DIOGO CONSTÂNCIO

FIGURA 2 - PEDRO ARAÚJO

Resumo

Este relatório complementa o Projecto de Linguagem Orientada a Objectos no âmbito da U.C. Laboratórios de Informática III, no qual se pretende recriar o Projecto Imperativo realizado previamente.

As fases de desenvolvimento do Projecto foram essencialmente as mesmas planeadas em C, com um grande cuidado no planeamento e desenvolvimento das estruturas de dados, de modo a não obrigar a travessias/tratamentos das estruturas no momento de resposta a *Queries*.

Estando agora num patamar de mais alto nível, questões como boa gestão de memória e portabilidade de código saem da nossa responsabilidade, facilitando assim o desenvolvimento, permitindo-nos focar no corpo do projecto sem grande preocupação a detalhes como apontadores e alocações/libertação de memória.

Analogamente ao Projecto Imperativo, o programa conta com as seguintes estruturas de dados:

GestHiper: Main, faz a ligação com a classe *HiperMercado* tendo ainda código de tratamento de *input*/*output*.

HiperMercado: Classe composta por instâncias dos Catálogos, Contabilidade e Compras, fazendo de mediador entre o Main(*GestHiper*) e estas.

Estruturas Básicas: Classes de utilidade genérica utilizadas pela Contabilidade, Compras, e mesmo Main(Estruturas de apoio a resultados).

Introdução

Estrutura da Aplicação

Naturalmente, o Projecto tem ainda modularidade em mente, não obrigando a este tipo de utilização específica como aqui apresentado, a classe *HiperMercado* e todas as suas sub-classes pertencem a um *package* com o mesmo nome, este poderia ser utilizado noutros contextos que não linha-de-comandos, como *GUI*, *Web,* ou apenas até para efeitos de estatística, sem comunicação com o utilizador.

Houve ainda uma preocupação com persistência de dados, de modos a não obrigar à leitura dos ficheiros base de Catálogos e Compras sempre que se corria o executável. Para se conseguir este funcionamento foi implementada a interface *Serializable* em todas as classes da *package Hiper-Mercado*, fazendo de seguida uso da API *ObjectInput/OutputStream*. No entanto esta funcionalidade apresentava erros no momento de leitura do ficheiro-objecto, não sendo possível resolver os problemas a tempo da entrega.

O Main (*GestHiper*) instância uma cópia de *HiperMercado*, tendo assim acesso não-directo às classes de Catálogos, Contabilidade e Compras, trata ainda de receber o *input* do utilizador e de apresentar resultados de forma legível. É também o ponto onde excepções atiradas por classes níveis abaixo são apanhadas.

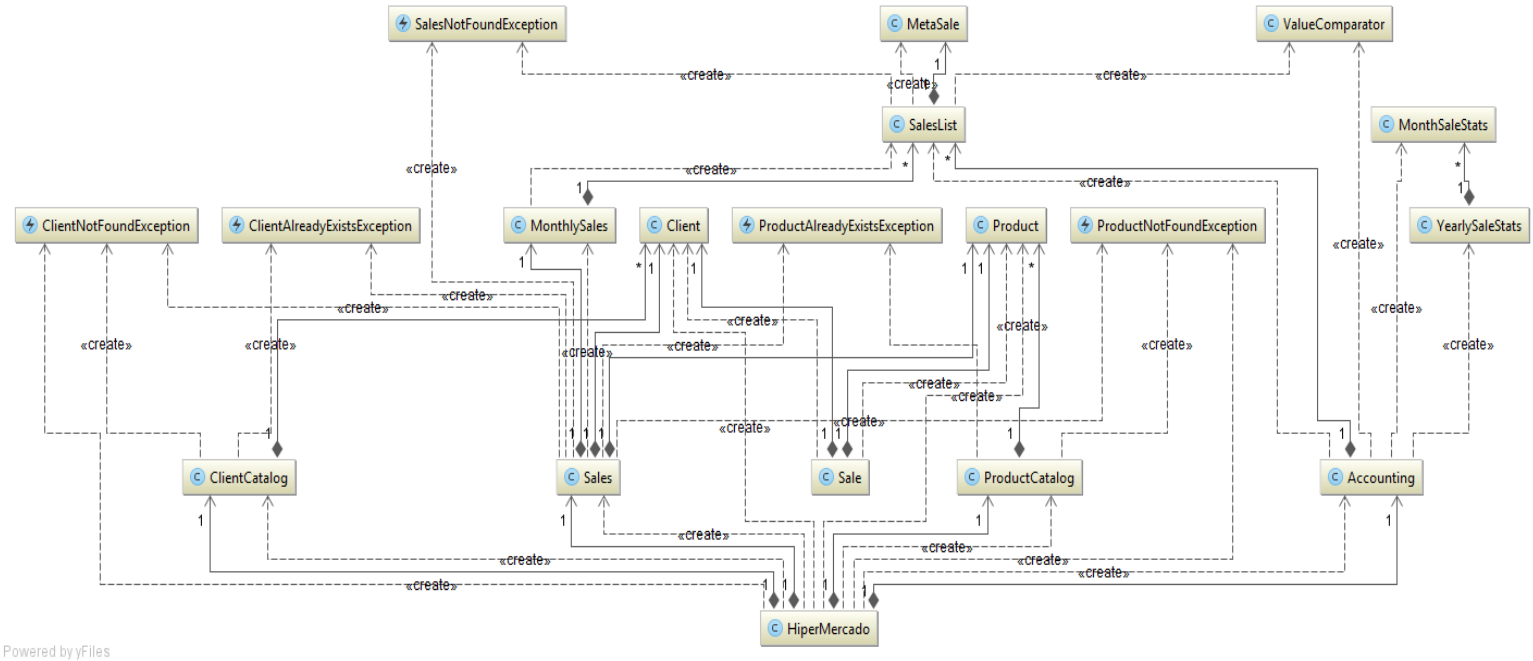

FIGURA 3 – DIAGRAMA DE HIPERMERCADO

HiperMercado

Nesta secção são tratadas as principais sub-classes utilizadas pela classe HiperMercado, sendo esta última apenas a API entre uma utilização e as classes fundamentais que seguram o funcionamento.

Menciona-se também os módulos básicos utilizados por estas classes, sendo eles como apresentados no diagrama anterior, *MonthlySales*, *SalesList*, *MetaSale*, e *YearlySaleStats*. A classe *Sale* é utilizada apenas no momento de inserção, segurando os valores base de uma compra como lida do ficheiro, sendo depois descompactada numa *MetaSale*, o funcionamento da qual será detalhado de seguida.

Para esta implementação do projecto GestHiper decidimos não guardar necessariamente a informação de uma compra específica que um cliente faz, mas sim a estatística resultante das compras que um cliente/produto faz num dado mês, obtendo-se assim estes valores em forma de somatório, guardados na classe *MetaSale*.

Esta decisão reflectiu-se num ganho de performance uma vez que não existe agora necessidade de iterar por todas as compras de um cliente, abrindo também um leque de novas relações entre produtos e clientes que não têm que ser obtidas forçosamente, permitindo-nos responder em tempo constante a questões como *de quem para quantos/quais e quando.*

Houve ainda uma preocupação com boas prácticas de programação em Java e dada a utilização da *Java Collections Framework*, pelo que todas as classes do projecto implementam o método *hashCode*, assim como *clone*, com algoritmos que asseguram a unicidade de um objecto.

A nível de ordenação, as classes relevantes para este efeito são *MonthlySales, MetaSale* e *SalesList.* Como tal estas classes implementam a interface *Comparable*, e por conseguinte o método *CompareTo*, este comportamento verifica-se no momento de ordenação das estruturas, através da utilização de *TreeMaps*, e *Comparators,* que fazem uso de uma classe de uso genérico de nome *ValueComparator.*

Catálogo de Produtos/Clientes

Verifica-se novamente que o funcionamento dos Catálogos são em tudo bastante semelhantes, como tal surgiu um interesse que tinha já aparecido no projecto de C, de ter apenas uma implementação de Catálogos, havendo portanto uma única classe de nome *Catalog*, face a implementação que se verifica aqui, de *ClientCatalog* e *ProductCatalog*.

Foram estudadas as metodologias que permitiram este funcionamento, este traduzir-se-ia em ter uma interface Pai como *Identifiable*, que seria mais tarde implementada pelas classes Produto e Cliente. Desta forma a classe única de catálogo seria agora por exemplo *ArrayList<Identifiable>*.

Compreendemos no entanto a necessidade de separar os dois por motivos de compatibilidade de futura, uma vez que mais tarde se poderá escrever novos funcionamentos para um Produto, ou mesmo para o próprio Catálogo de Produtos, e vice-versa para o Cliente.

Esse interesse futuro poderá tratar-se da necessidade de associar a um Cliente informações de facturação e contacto, no caso do Produto informações relativas preços base, escalões e mesmo outros dados de Marca. Estes exemplos básicos demonstram como uma única implementação de Catálogo que deveria agora albergar dois tipos de informações fundamentalmente diferentes passaria a necessitar de um desenho extremamente genérico, o que se traduziria em código bastante ambíguo.

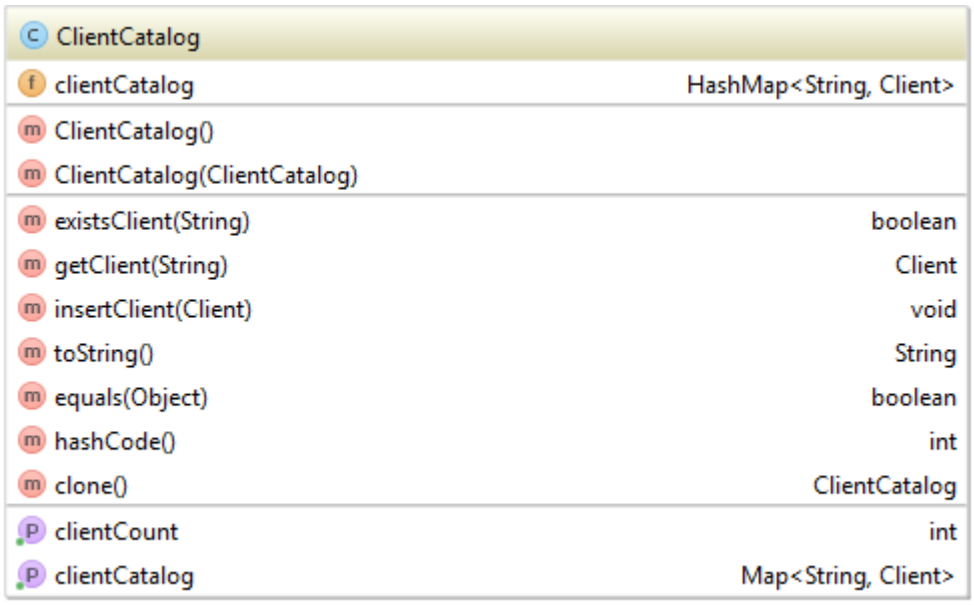

FIGURA 4 – DIAGRAMA DE CLASSE DE CATÁLOGO DE CLIENTES

Estrutura de Dados

Para representar e permitir a realização eficiente das operações relevantes foi escolhido um *HashMap* para guardar os Clientes/Produtos.

A razão pelo qual se fez esta escolha ao invés de algo como um *TreeMap* reside na frequência dos tipos de operação. Como inserções e procuras são muito mais frequentes num cenário de utilização normal que a listagem ordenada, a utilização de um *HashMap* torna estas operações mais rápidas, à custa de ordenação um pouco mais lenta.

Esta consequência não provoca no entanto grande impacto, uma vez que este enunciado não tem *queries* que façam apenas uso dos Catálogos, o único papel dos catálogos nesta implementação é a de registo de Clientes/Produtos válidos, para que no momento de leitura de compras se se verifique se os recipientes/originadores da compra são de facto dados válidos.

Tempos de Inserção	**Produtos**	**Clientes**
HashMap	1.210823268	0.05206419
TreeMap	1.472288765	0.065414783
Diferença	- 17.7 %	- 20.4 %

Contabilidade

Contabilidade contém a informação respectiva a Productos para Clientes, e vice-versa, ordenada por meses, permitindo-nos responder a perguntas como *quantos* e *quando*.

Essencialmente a informação que aqui consta está também presente na classe Compras, tanto mais evidenciado pela utilização das mesmas estruturas básicas utilizadas pela Compras, como *SalesList* e *MetaSale*.

A diferença baseia-se no tipo de utilização que é dada a estas estruturas, pelo que no módulo de Compras há uma necessidade albergar mais informação, e de estabelecer relações mais específicas entre Produtos/Clientes.

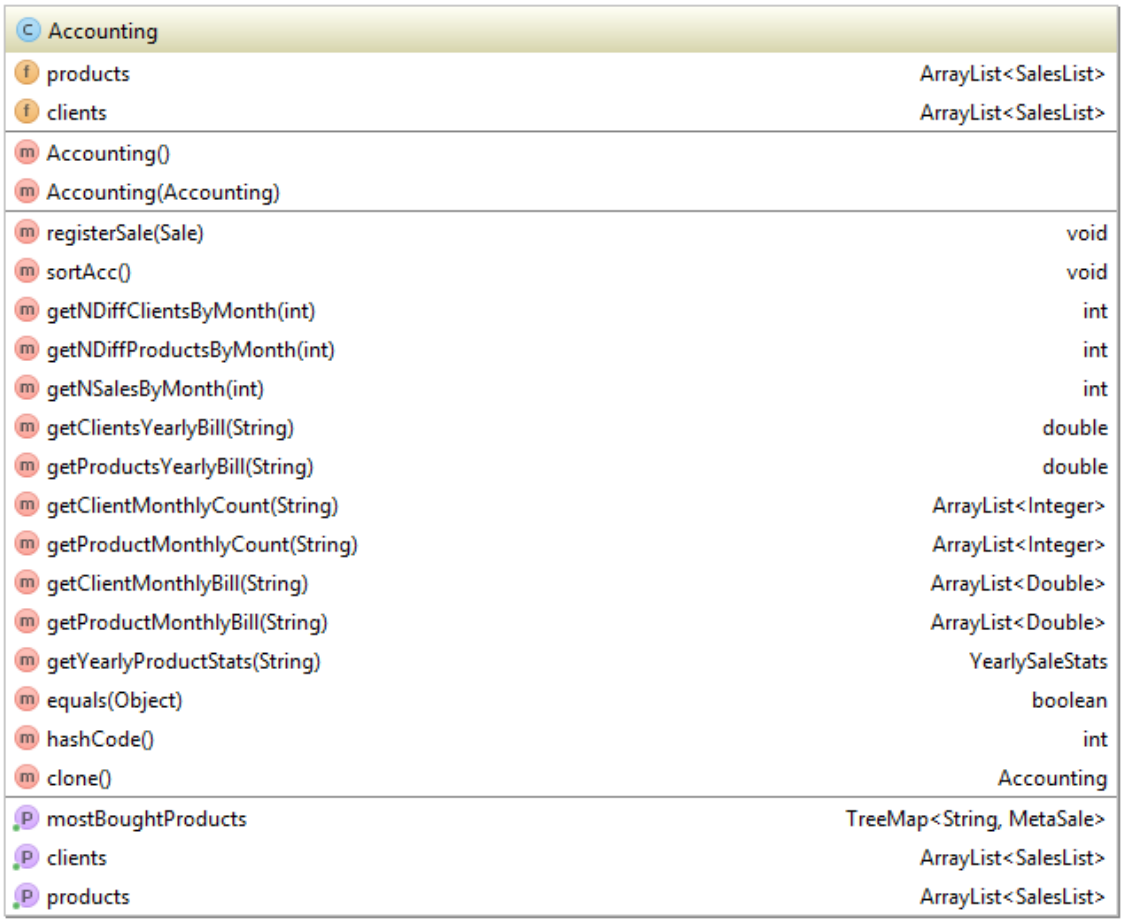

FIGURA 5 – DIAGRAMA DE CLASSE DE CONTABILIDADE

Estrutura de Dados

O desenho da estrutura aqui implementada permite aceder a resultados em tempos constantes, não obrigando a fazer intersecções entre classes básicas em níveis mais abaixo.

Grande parte deste funcionamento provém da base *SalesList* utilizadas nos *ArrayList* de meses, como se pode observar no diagrama seguinte:

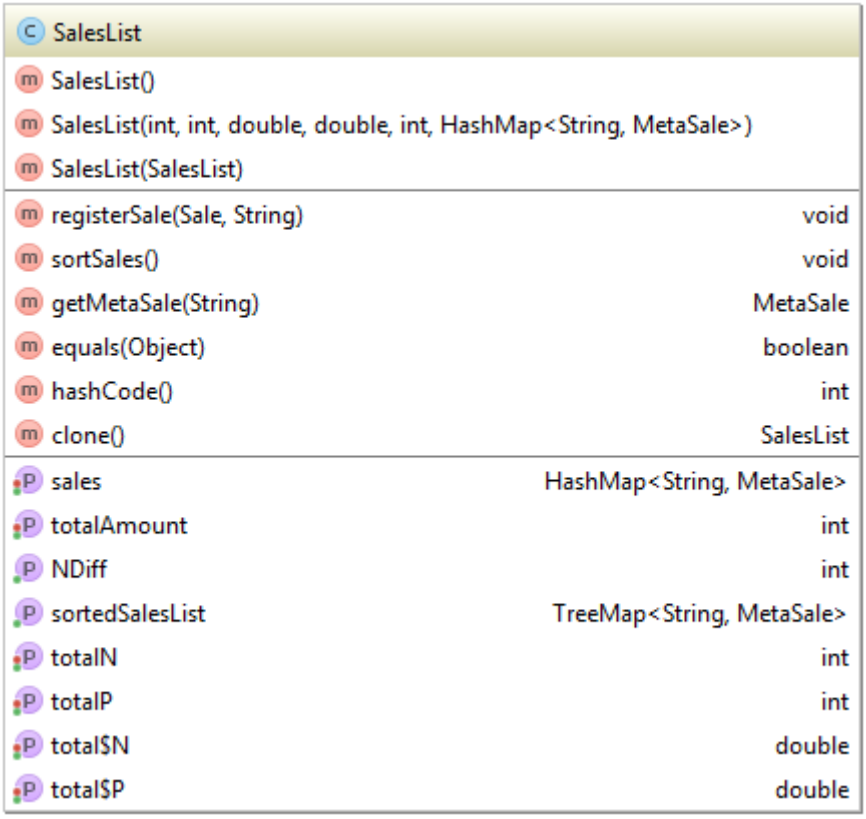

FIGURA 6 – DIAGRAMA DE CLASSE DE SALESLIST

Para além dos somatórios representados como *TotalN*, *totalAmount,* tem-se ainda um *Map* que relaciona Clientes/Produtos com as suas compras, o que permite por exemplo responder à questão de quantos Clientes compraram um determinado Produto, e quanto gastaram.

Estas implementam também uma forma de ordenação como pedida no enunciado, de forma decrescente em relação a totais, e segundamente de forma alfabética do código do Produto/Cliente, este funcionamento será explicado em detalhe em Compras.

Compras

Este módulo trata de responder a questões de *quem* para *quais (e* consequentemente, *quantos*).

Tendo em conta o enunciado específico do projecto, o módulo Compras é utilizado para practicamente todas as *Queries* solicitadadas, havendo apenas um caso em que se usa exclusivamente a Contabilidade.

Os resultados compilados por esta estrutura através da sua API devolvem *LinkedLists,* isto porque tratam-se das colecções mais simples que preservam ordem de inserção, guardar resultados num *TreeSet* não faria sentido uma vez que não existe necessidade de garantir unicidade de um valor nestes resultados.

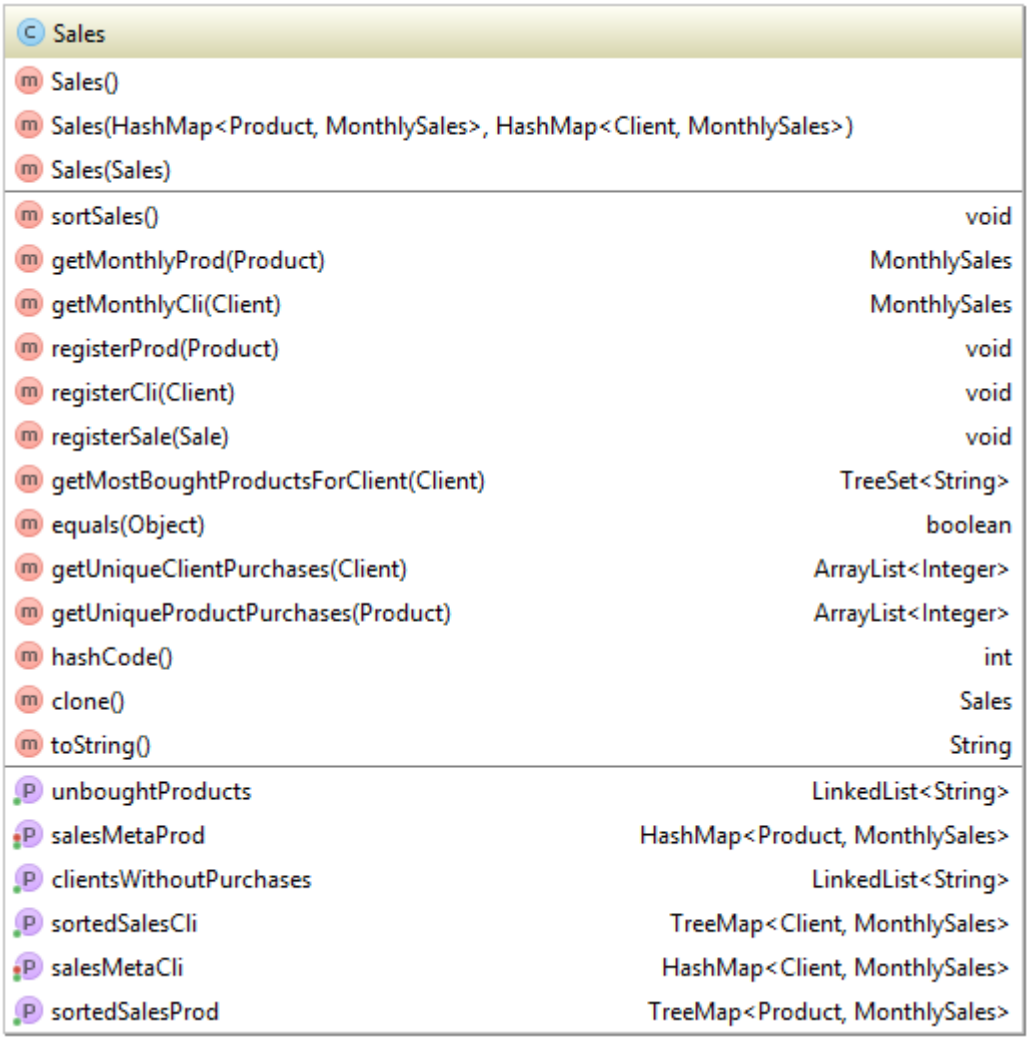

FIGURA 7 – DIAGRAMA DE CLASSE DE COMPRAS

Estrutura de Dados

Na base desta classe encontram-se dois *Map*s, de Clientes/Produtos e as suas vendas mensais. Inicialmente estes não tem ordem, e são populados tratando-os como *HashMap*, o motivo por trás desta decisão é simples, uma vez que para operações básicas o *HashMap* responde em tempo constante, enquanto que o *TreeMap* o faz em *log(n).*

Outro factor que influenciou esta decisão foi a ordenação por defeito do *TreeMap*, obrigando-o a fazê-la pelas chaves de um elemento, e não pelo valor associado a essa chave. Como tal, e de acordo com esta implementação de *Map<Product, MonthlySales>*, isto significaria que quer o Produto quer o Cliente passariam a ter que implementar a interface *Comparable*, o que não faria sentido uma vez que um Produto/Cliente só por si não tem base de comparação com outros, a não ser por ordem alfabética de código.

O processo de inicialização tem portanto um comportamento semelhante à do projecto de C, primeiro inserindo as compras lidas do ficheiro em cada um dos *Maps,* respectivamente em clientes e produtos, e de seguida, terminada a fase de inserção, executa-se o método que procede a ordenar ambos os *Maps*¸ assinalando também a *flag* de ordenação.

A ordenação ocorre fazendo uso da classe *ValueComparator,* nota-se agora a vantagem da implementação genérica desta classe, sendo utilizadas por várias outras. Começa-se por inicializar uma nova instância desta classe com o *Map* que se pretende ordenar, para servir de base de comparação, de seguida atribui-se a classe *TreeMap* aos *Maps* simples, colocando finalmente todos os valores dos anteriores *HashMap* nos novos *TreeMaps*. Efectivamente este processo de ordenação trata-se de apenas três linhas de código.

Medição

Através do uso da biblioteca *Crono* foi possível fazer testes de tempos de execução de diferentes aspectos do programa, desde o *parsing* dos ficheiros, à ordenação e execução de uma das *Queries*.

Mostra-se de seguida portanto várias tabelas com os tempos obtidos em determinados cenários, fazendo uso de colecções diferentes, pretende-se justificar as escolhas implementadas no projecto, representadas como *Default* na segunda tabela.

Os testes aqui realizados tiveram como base o ficheiro de compras com 1 Milhão de registos, de modos a se verificar um maior impacto nos tempos.

Processador: Intel Core i5-4210U CPU @ 1.7GHz
Memória: 8GB DDR3L Single Channel @ 1600MHz
Disco: 500GB SSHD

Leitura de Ficheiros	c/ Parsing	s/ Parsing
BufferedReader	0.473503986	0.289277463
Scanner	2.370217522	0.580111194
Diferença	- 80 %	- 50.1 %

Testes	Leitura	Ord.Contab	Ord. Compras	Q6
Default	3.004718366	0.840147105	0.645363934	0.001121674
HashMap	3.015905203	0.972233488	0.64527508	0.001348942
TreeMap	3.958900073	3.067595208	1.4931223	0.001131352
Vector	3.130533046	0.962970653	0.595329902	0.00141287

Os resultados aqui obtidos demonstram em particular a questão que se levantou na secção anterior, relativamente à escolha de uma ordenação posterior à inserção. Utilizando apenas *TreeMap* os tempos de inserção aumenta em 24%, para a ordenação (que terá que ser feita de qualquer das maneiras, uma vez que a inserção default ordena por chave) o aumento é mais considerável, 72.6% para a Contabilidade, 56.7% para a Compras.

Conclusões

Terminado o desenvolvimento do Projecto podemos fazer uma apreciação positiva do desenho escolhido das estruturas, novamente grande parte do tempo dedicado ao projecto foi gasto no planeamento das estruturas e do funcionamento geral da aplicação.

Infelizmente devido a uma má gestão de tempo não foi possível responder a todos os pontos de avaliação, mas apesar disto conseguiu-se ainda escrever código eficiente e genérico permitindo ampliar a utilização dada às bibliotecas aqui criadas.

Houve ainda um interesse por parte dos membros do grupo de comparar os resultados de medição obtidos no Projecto Imperativo com os conseguidos agora, sendo que o único ponto comparável será a leitura e inserção dos valores de Compras e Catálogos nas suas respectivas estruturas, como tal repara-se que se obteve um ganho de performance de 35.2% no Projecto OO, este ganho é compreensível tendo em mente as optimizações base fornecidas pelo próprio JDK e JRE.