

**Universidade do Minho**

Escola de Engenharia

**Unidade Curricular:**

**Laboratórios de Informática III**

Ano Lectivo de 2014/2015

Janeiro, 2015

**Relatório de Desenvolvimento de Projecto**

**GestHiper - Java**

**70719 Diogo Constâncio,**

**70513 Pedro Araújo**

# Índice

[Índice 2](#_Toc422690914)

[Índice de Figuras 3](#_Toc422690915)

[Fotos 4](#_Toc422690916)

[Resumo 5](#_Toc422690917)

[Introdução 6](#_Toc422690918)

[Estrutura da Aplicação 6](#_Toc422690919)

[HiperMercado 7](#_Toc422690920)

[Catálogo de Produtos/Clientes 8](#_Toc422690921)

[Contabilidade 10](#_Toc422690922)

[Compras 12](#_Toc422690923)

[Profiling 14](#_Toc422690924)

[Conclusões 15](#_Toc422690925)

# Índice de Figuras

Figura 1 – Diogo Constâncio 4

Figura 2 - Pedro Araújo 4

Figura 3 – Diagrama de HiperMercado 6

Figura 5 – Diagrama de Classe de Catálogo de Clientes 8

Figura 6 – Diagrama de Classe de Contabilidade 10

Figura 6 – Diagrama de Classe de SalesList 11

Figura 7 – Diagrama de Classe de Compras 12

# 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* *HiperMercado*, 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.

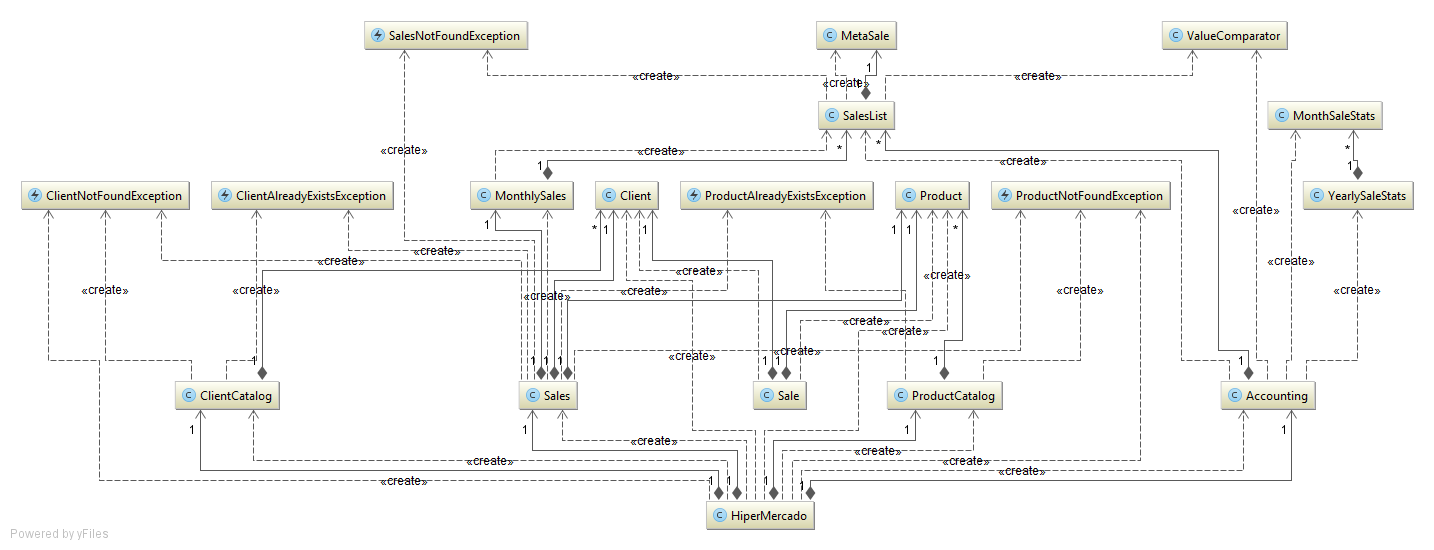
OMain (*GestHiper*) instância uma cópia de *HiperMercado*, tendo assim acesso não-directo às classes de Catálogos, Contabilidade e Compras, trata ainda de receber o *input* do utilizador e de apresentar resultados de forma legível. É também o ponto onde excepções atiradas por classes níveis abaixo são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

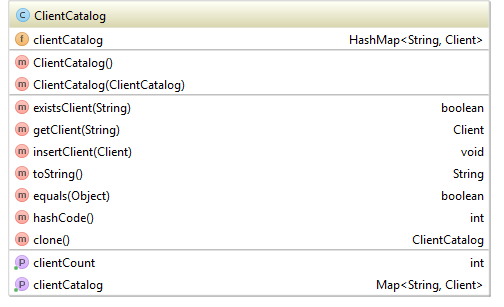


Figura 4 – Diagrama de Classe de Catálogo de Clientes

#### 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 |
| Ganhos | **-** 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.

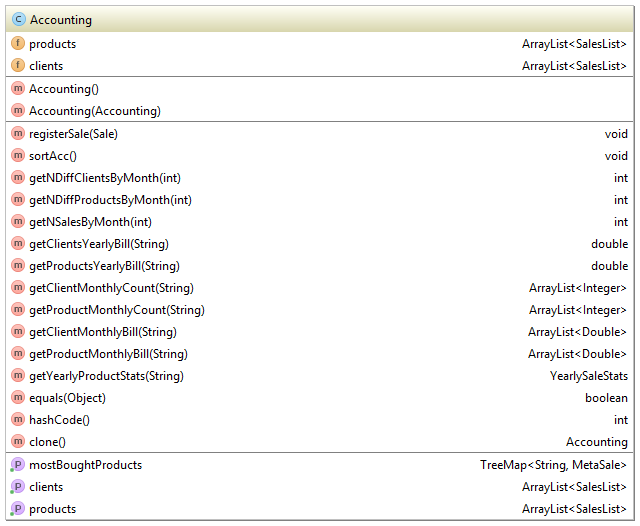


Figura 5 – Diagrama de Classe de Contabilidade

#### 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:

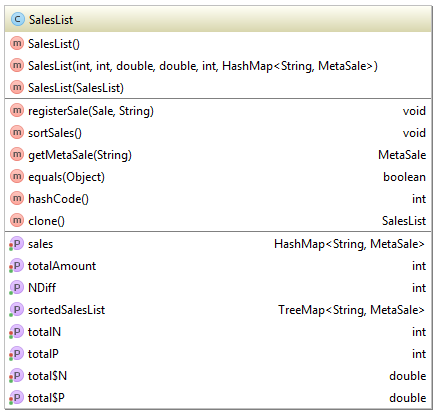


Figura 6 – Diagrama de Classe de SalesList

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 estabelecer a tabela relacional entre Produtos e Clientes, e vice-versa, relativamente a compras.

Embora o seu desenho seja bastante parecido com o da Contabilidade, este difere na natureza dos dados apontados por um Cliente/Produto, e na capacidade de relacionar directamente Produtos/Clientes com as compras que dizem respeito a estes.

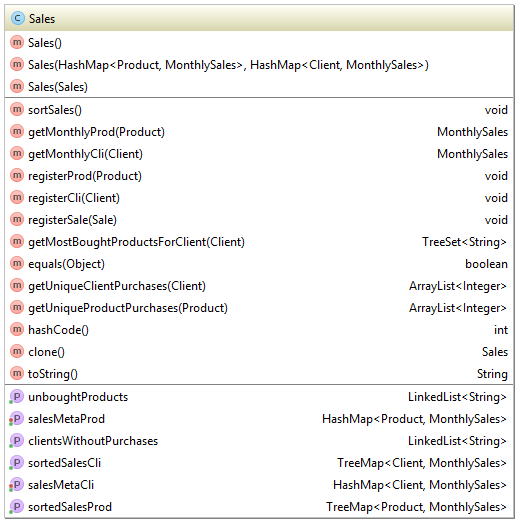


Figura 7 – Diagrama de Classe de Compras

#### Estrutura de Dados

A estrutura de dados usada é muito semelhante à utilizada no módulo de Contabilidade, diferindo no conteúdo de cada entrada, sendo também ordenado da mesma forma que os registos na Contabilidade, de forma a responder a *queries* com as mesmas características.

Cada entrada nos *arrays* tem a si associado a contagem total de unidades vendidas/compradas, seguindo-se do código do elemento correspondente a esta entrada, e finalmente um *array* que ordena, por mês, a contagem de compras associadas a este elemento.

As entradas neste módulo contém ainda 12 *arrays* de registos, em que cada entrada aponta para uma compra realizada no respectivo mês. Foi feita a decisão de manter intactos os dados pertinentes a cada venda por forma a permitir extensibilidade apesar de um maior uso de memoria, permitindo assim, se necessário, adaptar o módulo a novas necessidades.

A inserção segue a mesma linha que a estrutura de Contabilidade, ‘incrementando’ os valores de uma entrada com a Compra a ser processada actualmente. Este processo trata de incrementar as unidades desta entrada com as unidades da Compra, seguindo-se do registo da Compra no *array* geral de *Compras*, e a consequente associação desta Compra, já colocada na estrutura *Compras*, ao seu Produto e Cliente respectivo.

A ordenação executada nestas estruturas é 100% idêntica à processada na estrutura Contabilidade, sendo que a variável ordenante trata-se do somatório de unidades ao longo do ano, aqui disponível sob a forma de uma única variável-total.

# Profiling

Para efeitos de comparação e como forma simplística de avaliação de *performance*, apresenta-se de seguida uma tabela com tempos de execução para cada *query*. Os seguintes factores devem ser considerados, relativamente aos tempos apresentados, método de análise do tempo de execução e configuração do computador utilizado. De se notar, no entanto, que apesar destas considerações, os tempos apresentados são sujeitos a circunstâncias inalteráveis das situações em que os testes foram realizados: como outros processos em execução, *caching* do disco, etc…

O método empregue para levantar estes tempos faz uso da biblioteca *time.h*. Por motivos de portabilidade de código foi utilizada a função *clock()* face a *time()*, uma vez que a última difere no valor de retorno em plataformas diferentes.

A função *clock()* devolve a contagem de ciclos, os tempos são obtidos tendo em conta a constante *CLOCKS\_PER\_SEC*, declarada na biblioteca *time.h*, uma vez que esta terá em conta a plataforma utilizada, garantindo a portabilidade do código.

Infelizmente e apesar desta atenção, a precisão dos tempos calculados só abrange os milissegundos (sob a forma de número inteiro), sendo que se uma operação analisada não demorar mais que um milissegundo, o tempo devolvido é 0, pelo que a única certeza neste caso é *<1ms*.

O computador-portátil utilizado para calcular estes tempos possui as seguintes características relevantes:

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Query #** | **Compras 500k** | **Compras 1M** | **Compras 3M** |
| 1 | 12.72 s | 22.15 s | 56.27 s |
| 2 | 7 ms | 8 ms | 9 ms |
| 3 | <1 ms | <1 ms | <1 ms |
| 4 | 28 ms | 17 ms | <1 ms |
| 5 | <1 ms | <1 ms | 1 ms |
| 6 | 2 ms | 3 ms | 3 ms |
| 7 | 7 ms | 6 ms | 6 ms |
| 8 | <1 ms | <1 ms | <1 ms |
| 9 | <1 ms | <1 ms | 1 ms |
| 10 | 6 ms | 11 ms | 11 ms |
| 11 | 9 ms | 9 ms | 9 ms |
| 12 | 30 ms | 66 ms | 106 ms |
| 13 | <1 ms | <1 ms | 5 ms |
| 14 | 14 ms | 4 ms | <1 ms |

# Conclusões

Terminado o desenvolvimento à data de entrega, podemos fazer uma apreciação bastante positiva do projecto concluído, tratando-se de uma fiel representação das aptidões dos membros de equipa.

Como referido em partes anteriores do relatório, algumas optimizações possíveis não foram implementadas por forma a permitir uma maior extensibilidade. Mantemos que esta foi uma decisão acertada dadas as características do *hardware* moderno, que nos permitem ser um pouco menos conservativos com uso de memória ou tempo de CPU por forma a facilitar modificações/reutilização futura.

Todos os pontos de avaliação foram considerados e atingidos, pelo que o produto final se revela como uma aplicação robusta e bastante eficiente, como evidenciado pelos tempos de execução (tendo em conta os factores mencionados no *Profiling)*.