Universidade do Minho
Departamento de Informática

Mestrado Integrado em Engenharia Informática

Laboratórios de Informática 3

Segundo projeto - Java

A82238 João Gomes
A80376 Pedro Pereira

Braga
2020

Conteúdo

1 Introdução

Este segundo projeto é em muitos aspetos semelhante ao primeiro. A maior diferença encontra-se nos requisitos pedidos, sendo estes em quantidade muito mais elevada do que na primeira parte do projeto. Achamos que este aumento da quantidade de requisitos é justificada, dada a facilidade que Java fornece para realizar outras partes do trabalho.

2 Interpretação do Enunciado

Após a leitura e interpretação do enunciado, verificamos que as estruturas de dados que usamos para a primeira parte do projeto não vão em muito mudar na implementação desta segunda parte do projeto. As queries pedidas foram também divididas em dois grupos diferentes: um grupo de queries estatísticas, onde serão apresentados dados relacionados com os ficheiros lidos, e um grupo de queries interativas, onde o utilizar escolhe qual o tipo de dados a ver.

Uma grande componente deste projeto foca-se na realização de testes de performance e medição de tempos, para comparar diferentes alternativas e diferentes soluções possíveis.

Foi-nos também pedido para implementar um mecanismo que permitisse ao utilizar guardar o estado das estruturas de dados num *.dat* file, e permitisse também carregar os dados na estrutura a partir desse ficheiro, ao invês ter de ler os ficheiros de dados.

3 Descrição do Espaço de Desenho

3.1 Modularidade

Um dos conceitos mais importantes abordados neste Unidade Curricular foi o conceito de Modularidade, que consiste na separação do código fonte em vários ficheiros, normalmente agrupados por utilidade ou funcionalidade.

Ao contrário de C, Java é uma linguagem desenhada especificamente para suportar e facilitar modularidade, e portanto foi-nos bastante fácil aplicar este conceito ao nosso projeto.

Em java, estes módulos são chamados de *package* e permite-nos agrupar diferentes classes em grupos, consoante a sua funcionalidade. Temos então que as packages criadas por nós são:

- **Reader**: Package composta pelas classes *Reader* e *Parser* e pelas interfaces respetivas, responsáveis por fazer a leitura e parse dos códigos dos ficheiros fornecidos e de guardar a informação nas nossas estruturas de dados.
- **Modelo**: Package composta pelas classes *Catalogo*, *FaturaçãoGlobal*, *GestaoFilial*, *GestVendas* e *Vendas*. Estas são as classes que contêm as estruturas que irão guardar os dados associados ao ficheiros lidos, quer dos clientes, produtos ou do ficheiro de vendas.
- **View**: Package que apenas contém a classe *View*, responsável por apresentar ao utilizador os resultados das queries da melhor maneira possível.
- **Auxiliar**: Package que contém classes auxiliares ao projeto. Contém as classes *Input* e *Crono* fornecidas pela equipa docente. Contém também classes que nós achamos por bem criar, para facilitar a gestão da informação de algumas queries.
- **Package default**: Package default, que apenas contém o controlador e a classe *GestVendasAPPMVC*, que atua como a classe que junta todos as packages do projeto.

3.2 Estruturas de Dados

No que toca às estruturas de dados a usar, usamos uma abordagem muito parecido com a que usamos no primeiro projeto em C. Enquanto que no projeto de C fizemos uso da library glib, agora fizemos uso das diversas collections que java nos facilita.

- **Catalogo de Produtos e Clientes**: Para guardar a informação referente aos códigos dos produtos e dos clientes, decidimos usar a collection TreeSet para armazenar a informação. Esta foi uma escolha bastante direta e óbvia para nós, uma vez que TreeSet nos permite organizar os produtos por ordem alfabética sem grande esforço.

- **Vendas**: Estrutura auxiliar que apenas contém dois arrays. Um que terá 12 entradas, um para mês, e contabilizará as quantidades de vendas. O outro também terá 12 entradas, um para cada mês, e contabilizará a faturação total. Esta classe vai ser usada em conjunto com a faturação global e com a gestão filial.

- **Faturação Global**: Esta classe usada para guardar a informação relacionada com cada produto e todas as vendas e faturações a ele associados. Para tal, decidimos usar um HashMap. As keys são os códigos dos produtos e os Values, são uma instância diferente da classe Vendas, apresentado em cima.

- **Gestão de Vendas**: Esta é apenas uma estrutura que dá wrap a todas as outras estruturas, que resulta numa estrutura que dentro terá, uma estrutura de catalogo de produtos, uma de catalogo de clientes, uma de faturação global e estruturas de Gestão de Filial, uma para cada filial.

- **Gestão Filial**: Classe principal que contém guardada informação sobre todas as vendas, relacionando clientes com produtos. Usamos para isto um HashMap, que associa a cada código de cliente um outro HashMap. Este segundo HashMap, por sua vez associa um código de um produto a uma estrutura Vendas. Temos assim uma maneira fácil de dado um cliente e um produto, associar à estrutura Vendas que irá conter toda a informação das vendas realizadas entre este cliente e este produto.

4 Escolha Justificada da Solução Adotada

Em relação ao primeiro projeto em C, decidimos organizar a informação de uma maneira reduzida no Módulo da Gestão Filial. Enquanto que no projeto em C, tinhamos basicamente dois HashMaps que guardavam a mesma informação, um em função de clientes -¿ Produtos e outro em função de Produtos -¿ Clientes, neste projeto decidimos apenas ter informação relativa a Clientes -¿ Produtos.

Optámos por esta solução pois verificámos que os temos de loading dos dados para as estruturas estavam demasiados elevados, reduzindo assim para quase metade os tempos obtidos. No entanto, com esta solução os tempos de execução de certas queries é mais elevado, mas achamos que as vantagens pesam mais que as desvantagens.

5 Descrição da Estrutura do Projeto

A estrutura deste projeto foi feita tendo em conta o modelo MVC (Model View Controller).

Como já foi referido anteriormente, o nosso projeto encontra-se dividido em diferentes packages, cada um guardando as classes que possuem determinadas funcionalidades no contexto geral do trabalho.

As packages mais importantes são *Modelo* e *View*. Na package Modelo, encontram-se todas as classes que foram usadas para guardar a informação dos ficheiros em estruturas de dados escolhidas por nós. Encontram-se também as classes responsáveis pelo tratamentos dos dados e por dar resposta às queries pedidas pelo utilizador. Na package View, encontra-se apenas uma classe que é responsável por apresentar os resultados das queries ao utilizador da melhor forma possível. Como algumas das queries retornam resultados muito extensos, decidimos apresentar as respostas em forma de lista. Apresentamos 10 resultados de cada vez, e cabe ao utilizador decidir se quer ver os 10 próximos resultados, os 10 resultados anteriores ou procurar por um resultado em específico caso exista. Continuando segundo o MVC, implementamos também a classe *Controlador* que faz a ponte entre a View e o Modelo. O controlador pede ao Modelo o resultado das queries e depois passa o resultado à view, que depois será responsável de apresentar o resultado ao utilizador.

6 Testes de Desempenho

Um dos requisitos deste projeto era a realização de diversos testes de performance. Testes esses que abrangem a leitura sem parsing dos diferentes ficheiros, leitura com parsing, leitura com parsing e validação. Para realizar os testes de desempenho foi feito um for loop para cada teste que o iterava cinco vezes que depois foi corrido duas vezes que resulta num total de dez medições para cada teste individual. Embora não seja a melhor forma de testar porque testes sequenciais dão resultados ligeiramente diferentes do que testes isolados achamos que o volume acrescido de testes resultante da iteração através de um loop aumenta mais a qualidade dos dados do que a diferença ligeira entre testes decresce.

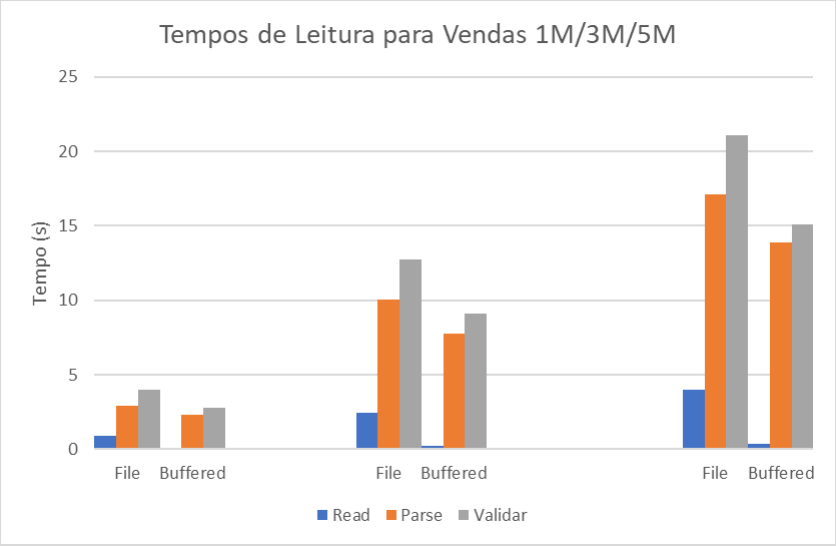

Figura 1: Tempos(s) medios de leitura parsing e validação dos tres ficheiros de vendas. Cada valor representa a media de dez valores

7 Interpretação

Podemos observar pelos resultados apresentados na Tabela 1 do anexo que o tempo de leitura, parsing e validação aumenta linearmente com o tamanho do ficheiro tanto para a leitura buffered como para a leitura normal. Para as operações que demoram mais a linearidade é mais evidante como por exemplo na validação dos três ficheiros:

$$File : 4.01 \approx \frac{12.74}{3} \approx \frac{21.11}{5} \tag{1}$$

$$Buffered : 2.81 \approx \frac{9.12}{3} \approx \frac{15.07}{5} \tag{2}$$

A diferença de tempo de leitura e a diferença de tempo de parsing entre testes normais e buffered tende a ser semelhante. Por outro lado, a validação aumenta a diferença de tempos o que se deve ao parsing adicional feito quando se tem de validar uma venda. Finalmente podemos constatar que a primeira medição feita em cada ciclo tende a ser mais demorada principalmente na leitura normal. Este fenómeno parece ser menos relevante para a leitura buffered talvez por causa dos tempos de leitura reduzidos.

8 Conclusão

Esta segunda fase do projeto permitiu-nos perceber que certas linguagens de programação estão equipadas para lidar com certos problemas melhor que outras. Java certamente fornece ao programador ferramentas que facilitam este tipo de projetos.

Neste projeto não tivemos de ter tanto cuidado com questões como Modularidade e Encapsulamento, uma vez que Java fornece esses conceitos à partida.

Comparando com a primeira parte do projeto em C, estamos muito satisfeitos com os resultados obtidos, uma vez que consideramos que os objetivos pretendidos foram cumpridos.

9 Anexo

	1M		3M		5M	
	file	buffered	file	buffered	file	buffered
ler	1.114804	0.086121	2.721329	0.272466	4.232593	0.340208
	0.862136	0.188038	2.319033	0.21642	3.873367	0.327484
	1.044515	0.099485	2.328304	0.219644	3.808225	0.51542
	0.932097	0.091604	2.316044	0.218654	3.872971	0.335198
	0.847874	0.091925	2.336642	0.216883	3.864234	0.33713
	1.162658	0.095238	2.903742	0.269155	4.364347	0.381714
	0.841288	0.085655	2.411675	0.22045	3.931088	0.355348
	0.806663	0.080016	2.421224	0.222961	3.926539	0.353936
	0.806179	0.080393	2.400431	0.220238	3.853679	0.349986
	0.798363	0.079636	2.413646	0.223133	3.87774	0.349839
parse	3.584948	2.585452	10.65691	8.02518	22.50071	14.49324
	2.943074	2.297971	9.555351	8.166226	16.96516	13.37559
	2.711169	2.159664	9.432018	7.875067	16.42343	14.54899
	2.783203	2.299435	9.751573	7.799933	14.82411	13.46512
	2.528148	2.276277	9.206483	7.886784	13.88475	13.25165
	3.502519	2.223818	11.79592	8.117012	21.45148	14.57225
	2.921994	2.067728	9.784635	7.774617	18.131	15.36175
	2.840671	2.634763	10.10153	7.371864	16.74623	13.43575
	2.538043	2.314285	11.24742	7.393432	16.00719	13.08836
	2.495894	2.201672	9.141816	7.44777	14.28778	13.07324
validar	4.176672	3.144615	13.04324	9.374533	21.01389	15.09871
	4.307735	2.724645	13.08085	9.193807	21.26273	15.17516
	4.136525	3.120526	13.33698	8.903009	21.02618	14.96293
	4.253577	2.706187	13.60219	8.912767	21.07761	15.10232
	3.808244	2.709958	12.50757	9.750374	21.50257	15.22289
	3.916246	2.938898	12.44324	8.694421	20.97114	14.88103
	3.740921	2.616124	12.48425	8.786462	20.7901	14.83665
	3.902808	2.931131	12.31029	9.201433	20.88895	14.95401
	4.130262	2.566394	12.28812	9.697065	20.96191	15.17375
	3.796759	2.657525	12.28144	8.670866	21.59033	15.32053

Tabela 1: Tempos de execução para os testes de performace