LI3: Trabalho Prático 2ª Parte
Stack Overflow - Java

Mestrado Integrado
em
Engenharia Informática

Grupo 69

Nome	Nº
Pedro Machado	a33524
Maurício Salgado	a71407
João Coutinho	a86272

12 de Junho de 2018

Conteúdo

1 Introdução

Esta segunda parte do trabalho propõe o mesmo desafio que a primeira (elaborada em C), mas desta vez em *Java*: uma linguagem orientada aos objectos. Como seria de esperar, a mudança de paradigma teria implicações a nível da estruturação e execução das interrogações pedidas. Relembrando a 1ª parte do trabalho em C, onde se tentou ter uma abordagem orientada aos objectos numa linguagem imperativa(com a definição de get's e set's), foi possível constatar uma simplificação a nível de execução de certas tarefas.

Ao longo deste relatório além de procurar justificar as decisões e abordagens usadas, estabelecem-se também comparações e conclusões em relação ao trabalho anterior. Procurando assim comparar a abordagem imperativa com uma orientada aos objectos, tirando conclusões sobre as diferenças.

2 Breve Descrição Geral

Depois de elaborado o trabalho em C havia já toda uma metodologia mental do que precisaríamos a nível de variáveis e de funções, o que poupou algum tempo. Tempo esse que foi usado a explorar novos conceitos e diferentes organizações de código, tentando tirar disso o máximo partido. Foi decidido tentar a abordagem **MVC** como sugerido pelos docentes, a qual foi possível constatar uma grande eficácia no código, como explicaremos mais à frente.

Model: Esta camada é constituida pelas estruturas de armazenamento de dados, e o seu tratamento(incialização, parse, clean). Será na nossa aplicação a classe Community e todas as classes directamente ligadas.

View: No nível da View foi criado um menu em modo texto onde o Utilizador pode efectuar as várias queries propostas, e visualizar os outputs das mesmas.

Controller: Faz a comunicação da View com o Model através das várias queries, e é implementado na Classe Controller.

3 Packages e Estrutura de Ficheiros

Foi adoptada a estrutura de packages implementada pelos docentes, e foram apenas recondicionados(segundo o tipo de funcionalidade ou categoria) os ficheiros dentro da estrutura.

engine: Neste package ficam todas as classes relativas ao nosso *Model*: Parser, Classes de dados, Excepções e Comparators.

li3: Além da classe *Main*, neste package ficam armazenadas as classes das componentes *View* e *Controller*.

common: Este package agrupa classes de utilidade, não relacionadas com as restantes.

4 Encapsulamento de dados

Todas as classes definidas têm variáveis de instância privadas, cujo acesso e alteração só poderá ser efectuado mediante *get's* e *set's*.

Em relação a **Clones** dado o contexto específico do programa (e também os dados retornados serem imutáveis), não faria sentido em muitos dos casos clonar objectos uma vez que a reduzida interacção com o utilizador asseguraria a protecção dos dados.

5 Componente Model

5.1 Classe Abstracta Post

Classe que representa um **Post**. Dado que um *post* será sempre uma questão ou uma resposta a uma questão, apenas faria sentido **Post** ser uma classe abstracta que agrega apenas as variáveis e os métodos comuns às duas subclasses, não sendo assim permitido criar instâncias deste tipo. Além dos métodos convencionais de uma classe (*getters*, *setters*, construtores), foram também criados métodos abstractos que obrigam as suas subclasses a os implementar.

```
13  public abstract class Post {
14      /** ID de um Post */
15      private long id;
16      /** Tipo de um Post */
17      private int type;
18      /** ID do autor */
19      private long id_user;
20      /** Score do Post */
21      private int score;
22      /** Data do Post */
23      private LocalDate date;
```

Figura 1: Variáveis de Instância da Classe **Post**

5.2 Classe Question

Subclasse da classe **Post**, esta classe especifica os *posts* do tipo 1, que são questões. Contém as variáveis de instância e os métodos convencionais deste tipo de *post*, e implementa os métodos abstratos de **Post**.

```
15  public class Question extends Post {
16      /** Título do Post */
17      private String title;
18      /** Número de Answers */
19      private int answers;
20      /** Lista de tags do Post */
21      private ArrayList<String> tags;
```

Figura 2: Variáveis de Instância da Classe **Question**

5.3 Classe Answer

Subclasse de **Post**, esta classe representa os posts do tipo 2, que são respostas às questões colocadas. Contém as variáveis de instância e os métodos convencionais deste tipo de *post*, e implementa os métodos abstratos de **Post**.

```
13  public class Answer extends Post {
14      /** ID do Post ao qual é resposta */
15      private long id_parent;
16      /** Número de comentários à resposta */
17      private int cmts;
```

Figura 3: Variáveis de Instância da Classe **Answer**

5.4 Classe User

Classe que representa um utilizador, guardando as suas variáveis de instância e os seus métodos convencionais. De notar a variável **posts**, uma lista com todos os *posts* feitos por um dado utilizador, que foi criada apenas para ser usada pela *query* 9, onde a sua existência melhora drasticamente os tempos de execução da mesma.

```
13  public class User {
14      /** Nome de um utilizador */
15      private String name;
16      /** Bio de um utilizador */
17      private String bio;
18      /** ID de um utilizador*/
19      private long id;
20      /** Reputação de um utilizador */
21      private int reputation;
22      /** Número de posts feitos por um utilizador */
23      private int postcount;
24      /** Lista dos posts feitos por um utilizador */
25      private List<Post> posts;
```

Figura 4: Variáveis da Classe **User**

5.5 Excepções

Para além das excepções usadas na abertura dos ficheiros, foram também implementadas excepções apropriadas ao nosso modelo. No caso de um *post* ou *user* não existirem foram criadas excepções, que permitem a execução normal do programa sempre que estas circunstâncias ocorrem.

Depois da análise das várias *queries*, achou-se as seguintes excepções seriam necessárias:

- **PostDNEException -** "Post Does Not Exist Exception", lançada quando uma busca pelo ID de um *Post* retorna NULL
- **UserDNEException -** "User Does Not Exist Exception", lançada quando uma busca pelo ID de um *User* retorna NULL
- **NotQuestionException -** Quando o input é suposto ser uma pergunta mas é fornecido o ID de uma resposta, o que daria problemas em algumas das queries.

5.6 Comparators

Dada a necessidade de ordenar resultados segundo vários critérios, foram criados vários comparators. Os mesmos foram criados conforme o contexto específico de cada *query*, ou do critério de ordenação do resultado.

Segue uma breve descrição dos comparators que se entendeu serem necessários:

- **AnswerScoreComparator -** Compara o *score* entre duas *answers*.
- **PostDateComparator -** Usado para comparar as datas de dois *posts*.
- **PostScoreComparator -** Compara o *score* entre *posts*.
- **QuestionNAnswersComparator -** Compara o número de *answers* entre *posts* ou *comments* de uma *question*.
- **UserCountComparator -** Compara o número de *posts* de cada utilizador.
- **UserRepComparator -** Compara a reputação de cada utilizador.

6 Classe Community

É a classe principal do trabalho que agrega todas as estruturas de dados e implementa as *queries*.

```
22  public class Community implements TADCommunity {
23      /** Map de Posts: usa como key o ID */
24      private Map<Long, Post> posts;
25      /** Map de Users: usa como key o ID */
26      private Map<Long, User> users;
27      /** Map de Tags: usa como key a string da Tag */
28      private Map<String, Tag> tags;
29      /** Map de Listas de Posts: usa como key a data do post */
30      private Map<LocalDate, ArrayList<Post>> dates;
```

Figura 5: Variáveis da Classe **Community**

Como dito anteriormente, a nível de estruturas não se fugiu muito ao conceito usado no trabalho anterior, assim como variáveis utilizados. Aproveitou-se unicamente para tentar simplificar e evitar erros cometidos anteriormente, tentando melhorar aspectos do trabalho anterior.

Uma vez que os algoritmos de resolução das *queries* já estavam de certa forma delineados, decidiu-se que o foco seria a simplicidade e eficiência do código. Assim sendo, optamos por falar da abordagem geral usada nas *queries* ao invés de descrever cada *query* individualmente.

Queries: Abordagem

Quisemos mudar completamente a abordagem às *queries*, deixando para trás os ciclos for e funções auxiliares tirando assim partido das potencialidades do *Java*.

- Foram usadas **Streams** sempre que possível, pois para além da elegância e eficiência do código, tornam a leitura e entendimento do mesmo bastante mais simples.
- Uma vez usadas **Streams** na maioria das queries, foi de máxima utilidade o uso também de **lambda expressions**, que permitiu o encadeamento das funções e resolver quase totalmente as queries numa só "linha"de código.
- Evitar o uso de funções auxiliares (como no trabalho prático anterior), tornando o código o mais simples possível.
- Foram acrescentadas excepções às queries que achamos necessário, para que, em caso de erro, não interrompessem o decorrer do programa.
- Uso de **Comparators** que permitiram ordenar facilmente os resultados segundo qualquer parâmetro.

Tudo o que foi descrito acima e o que tinhamos aprendido previamente sobre **streams** leva-nos a afirmar que este tipo de abordagem reduz as operações e variáveis usadas pelas *queries*, e produzem um código simples, eficiente e inteligível.

7 Componente View

Esta componente é unicamente responsável por imprimir *outputs* e menus, sendo a mesma controlada através da componente *Controller*. Neste trabalho a *View* corresponde à classe de nome **View**, contida no *package* li3 juntamente com a **Main** e o **Controller**.

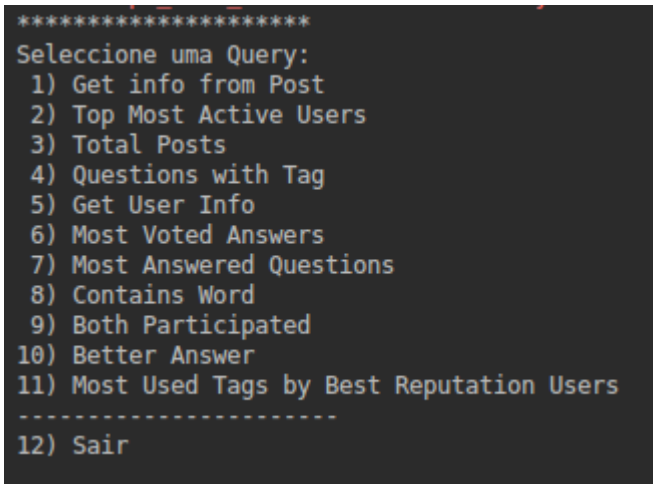

Figura 6: Menu de Interface do Utilizador

Durante a elaboração da Classe View, e respeitando a estruturação **MVC**, existiu certamente uma curva de aprendizagem a esta nova abordagem. Por comparação ao menu criado no trabalho elaborado em C verificamos que foi evitada muita da redundância, como por exemplo: quando pedimos ao utilizador para introduzir duas datas, é usada apenas uma função; sempre que são pedidos inputs ao utilizadores a mesma função é reutilizada várias vezes, e pode receber uma string como parâmetro (enviada pelo *controller*) referente ao que é pedido como *input*.

8 Componente Controller

É a componente que conhece a parte que deverá ser visível para o utilizador, que permite interagir com o modelo da estrutura, e que cria a ponte de ligação entre os dois.

```
22  public class Controller {
23      /** Instância da classe View referente à componente com o mesmo nome do MVC */
24      private View view;
25      /** Variável que armazena a componente Model do MVC */
26      private TADCommunity com;
27      /** Variável de registo dos resultados das queries */
28      private MyLog log;
29      /** Variável de registo dos tempos de execução do programa */
30      private MyLog logtime;
31      /** Variáveis de armazenamento de tempos de execução */
32      private long before, after;
```

Figura 7: Variáveis da Classe **Controller**

Esta classe é instanciada a partir da **Main**, e receberá como parâmetros o Model e a View após a inicialização dos mesmos. As queries serão invocadas a partir desta classe, que vai usando a classe View para obter os inputs e assim fazer os pedidos às queries, que uma vez obtidos serão mandados de volta à classe View para serem apresentados. Nesta classe são também geridos os logs de *output* e de tempos de execução das *queries*.

9 Desempenho

A flexibilidade do Java e o facto de termos declarado a maioria das estruturas como **Map** sem especificar qual o tipo de **Map** permitiu que fosse fácil intercalar entre as várias estruturas de dados.

Houve bastante surpresa com os resultados, pois a ideia do *Java* ser uma linguagem pesada e das *HashTables* serem muito mais rápidas do que *Trees* caiu um bocado por terra quando confrontados com os tempos de execução e *loading*.

9.1 Testes

De modo a tentar que os testes fossem o mais fidedignos possíveis, usamos o seguinte equipamento:

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 94
Model name:            Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz
Stepping:              3
CPU MHz:               899.945
CPU max MHz:           3500,0000
CPU min MHz:           800,0000
BogoMIPS:              5183.90
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              6144K
NUMA node0 CPU(s):     0-7
```

Figura 8: Características do PC de teste

Os testes foram executados 10 vezes consecutivas, e os tempos usados foram os tempos médios de execução.

9.2 Resultados

Neste 1º teste, testaram-se os tempos médios da queries usando HashMaps e TreeMaps

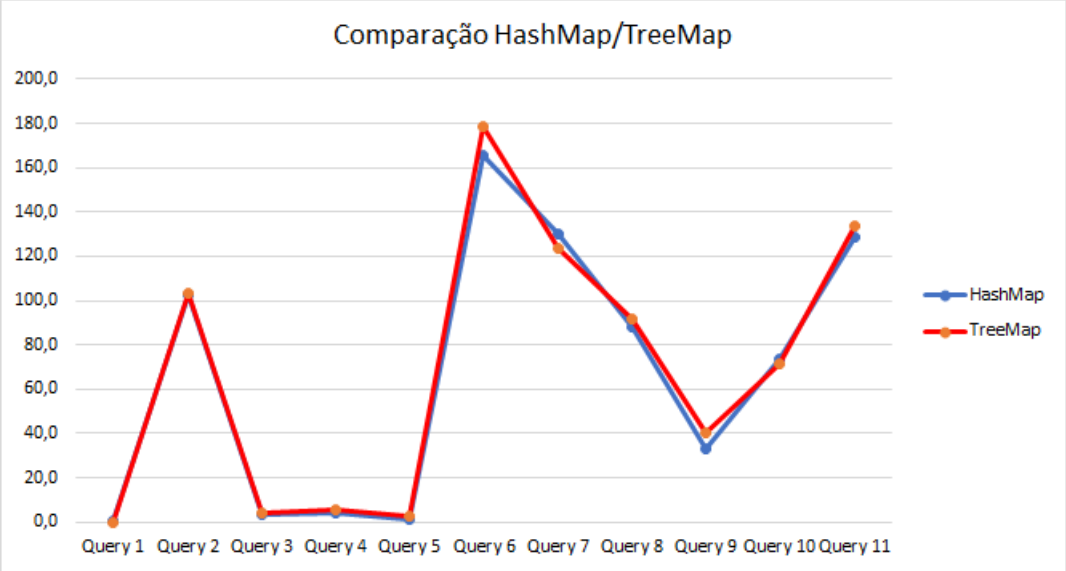

Figura 9: Gráfico de teste das queries, com tempo em milissegundos

Foi com grande espanto que se constatou que as medições foram tão idênticas, e que os *HashMaps* eram só marginalmente mais rápidos. Sendo *HashMap* a escolha inicial, acabou por vigorar essa decisão.

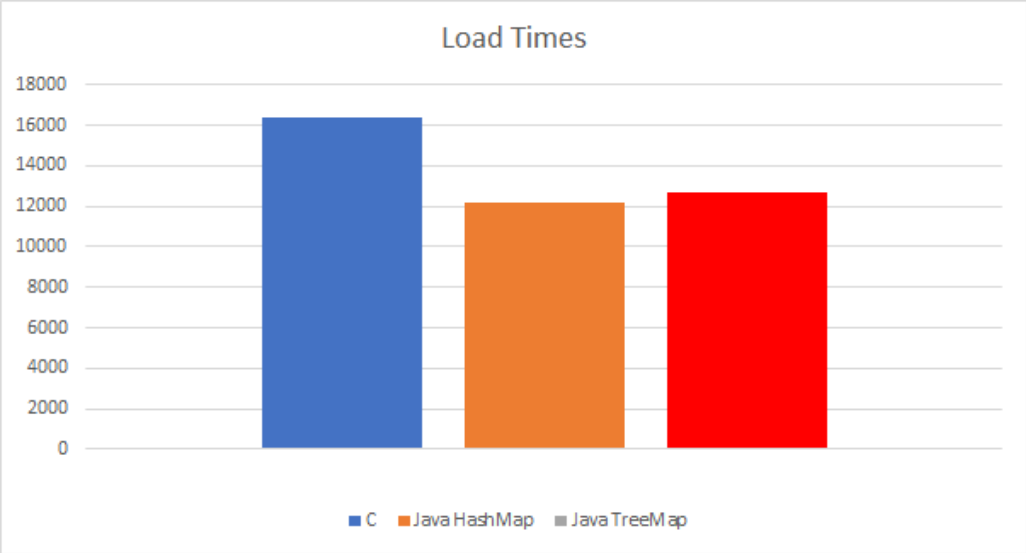

Figura 10: Gráfico dos tempos de load, tempo expresso em milissegundos

Mais uma vez uma surpresa os tempos de loading entre Java e *C*, onde pensariamos que C ganharia por uma larga margem. Não só foi mais lento, como é mais lento por uma margem considerável.

10 Notas Finais

10.1 Auto-Crítica

Mesmo muito satisfeitos com o resultado, haveria sempre aspectos a melhorar. Tendo sido este o primeiro contacto com **MVC** e até muito positivo, alguns aspectos da interacção com o utilizador poderiam ter sido melhorados ou feitos de forma diferente. Poderiam também ter sido criadas mais excepções de modo a impermeabilizar o programa em relação a erros.

10.2 Comparação com o trabalho anterior

É um bocado difícil pôr em palavras o quão importante a linguagem de programação é, dado o tipo de problema. Falando especificamente deste trabalho há um desfasamento de dificuldade e mesmo desempenho (como observado nos testes) que claramente elege o *Java* como o caminho a seguir neste tipo de problema. Desde a simplicidade do código ao uso de métodos já implementados tudo aponta a favor desta linguagem.

10.3 Eficiência

A eficiência foi certamente uma surpresa. Para quem acreditava que Java seria uma linguagem pesada e menos "limpa"que *C*, este trabalho veio desfazer todas as dúvidas. Depois de experimentadas as duas abordagens, a orientação aos objectos faz sem dúvida grande diferença.

10.4 Reutilização de código

Esta linguagem permitiu ter acesso a um vasto leque de funções já implementadas, o que simplificou bastante todo este desafio. O tempo dispendido ao criar construtores/*gets*/*sets* é rapidamente recuperado depois de toda a estrutura estar criada, pois existem métodos para todo o tipo funções e utilidades prontos a serem usados. A documentação e mesmo a internet mostrou-se uma ferramenta indispensável para a pesquisa dos métodos já existentes e verificar que há métodos para qualquer tipo de problema que surja.

10.5 Observações

Depois de realizar o mesmo trabalho em duas linguagens com abordagens distintas, e tendo em mente todas as diferenças observadas, dá para ter melhor noção do objectivo desta disciplina e também da importância de explorar dois paradigmas diferentes aplicados ao mesmo problema. Deu para observar as diferentes potencialidades de cada uma das linguagens e perceber que o contexto deverá definir a abordagem a seguir. Tudo isto vai permitir futuramente adequar melhores soluções a futuros problemas.

10.6 Apreciação Geral

Após a conclusão do trabalho, entendemos que a solução oferecida atingiu os objectivos a que nos propusemos: simplicidade, eficácia, e boa estruturação de código. Haverá sempre aspectos a melhorar, e certamente o trabalho seria feito de forma diferente após esta aprendizagem.

Pesando ambos os factores, o que foi elaborado *vs* aquilo que podia ser feito, consideramos a solução apresentada muito satisfatória segundo os nossos critérios.