Laboratórios de Informática III - Projecto em Java

Adriana Meireles (a82582), Eduardo Jorge Barbosa (a83344), Filipe Monteiro (a80229)

***Resumo*—O objectivo desta parte do trabalho foi re-implementar a solução desenvolvida em *C* num paradigma totalmente orientado a objectos em *Java*. Apesar de seguir a arquitectura anterior não é um factor que limite esta fase, sendo que *Java* proporciona um número enorme de estruturas já implementadas através das *Java Collections*.**

I. INTRODUÇÃO

Irá ser abordado neste relatório a arquitectura da nossa solução nomeadamente estruturas usadas, estratégias utilizadas nas *queries*, encapsulamento e o modelo *MVC*.

II. PARSER

Como o *parse* usado na primeira fase existe por *default* em *Java*, escolheu-se mais uma vez utilizar o parser do tipo *SAX*. É do notar a diferença entre a implementação em *C* e em *Java*. Utilizando uma implementação análoga do projecto original, o tempo de load aumentou cerca de **4 a 8 milisegundos**.

A. ALTERNATIVAS

Um *parser* alternativo que surgiu foi o *StAX*. Tal como o *Sax* este *parser* é orientado a eventos. A diferença entre os dois encontra-se no facto do *Sax* ser *push based* e o *StAX* ser *pull based*. De facto, o *StAX* pode ser visto como uma espécie de iterador que gera os eventos que o *parser handler* tem que lidar, já o *sax* recebe os eventos e lida com eles. Do que foi pesquisado *StAX* não oferece nenhuma vantagem sobre *sax* e sendo que para a parte de *C* já foi usado *sax* mantivemos.

III. MODULARIDADE

A. ENGINE

A nossa arquitetura segue este diagrama:

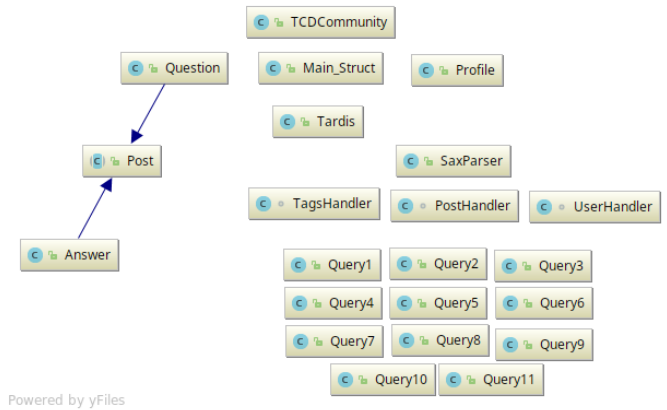

Figura 1. Classes principais

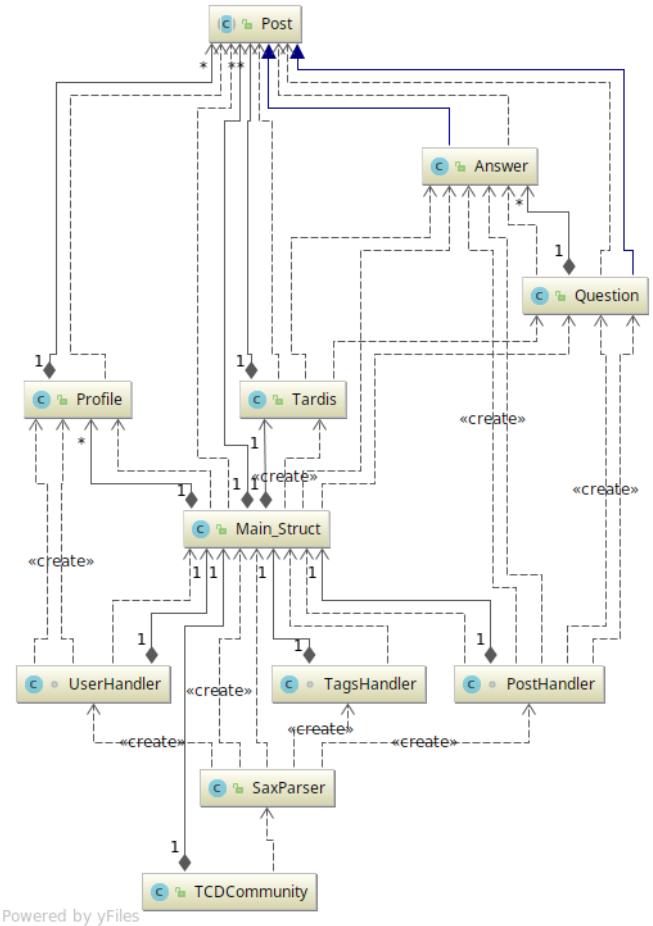

Figura 2. Diagrama de dependências

Os utilizadores são guardados na classe *Profile*. Questões e respostas são guardadas nas classes *Question* e *Answer* respectivamente que estendem a classe *Post*. Entra outras estruturas os *Posts* são guardados na *Tardis*. A classe *MainStruct* agrega todos os dados. Por fim, a classe *TCDCommunity* implementa a *interface* com as *queries*. Todas as *queries* foram resolvidas na sua respectiva classe sendo que cada classe apenas possui um método estático (que resolve a questão).

B. LI3 E MVC

Com base numa estruturação *MVC*, tentamos criar um menu simples, onde se pode testar cada *query* independentemente. Foi baseado no documento colocado na página da unidade curricular, com modificações para este contexto, assim como acréscimo de opções. Temos as seguintes classes:

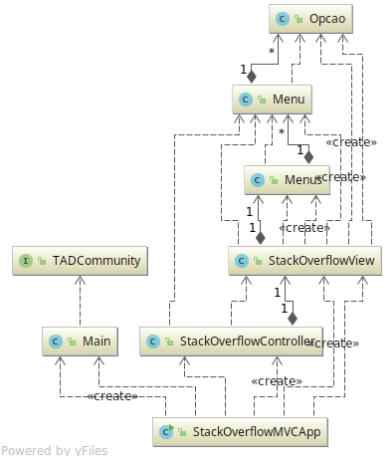

Figura 3. Package LI3

- *StackOverflowMVCApp*: *Main* do programa. Aqui é codificada a lógica de chamar as queries independentes, ou executar a *main* dada pelo corpo docente que corre todas as queries de uma vez, apresentando tempos e resultados;
- *StackOverflowView*: onde se encontram as views do programa (estáticas), os menus;
- *StackOverflowController*: controlador do flow do programa. Aqui fazem-se as decisões de menus a apresentar, chamadas das queries;
- *Opção,Menu,Menus*: opção, menu e conjunto de menus, respetivamente.

C. ENCAPSULAMENTO

De forma a garantir a preservação dos dados todos os *getters/metodos que retornem parte do estado interno* usados entre *packages* usam um *deep clone*.

IV. Classes e Estrutura

Em seguida irá ser feita uma descrição das estruturas utilizadas para efeitos de contextualização. Além do mais são **discutidas** certas decisões.

A. MAIN STRUCT

À semelhança do projecto em *C*, a estrutura principal é constituída por 3 *HashMaps*, uma para os perfis, outra para *posts* e outra para as *tags*, e pela *Tardis* que permite reaver os *Posts* organizados por qualquer critério.

1) OPTIMIZAÇÕES: Foi discutido a implementação de *ConcurrentHashMap* em vez das tradicionais. Infelizmente por falta de tempo não foi possível fazer um estudo desta *Class* para implementarmos logo não foi usada.

Alem disso onde são utilizados *TreeSets* foi pensado utilizar uma *heap* visto as inserções serem $O(1)$ em tempo amortizado. No entanto dado que previsavámos uma ordem mais rigorosa usamos *TreeSet* cujo tempo de inserção é $O(\log n)$.

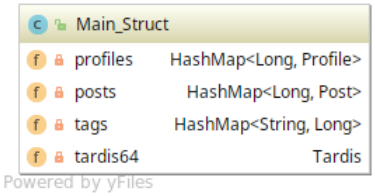

Figura 4. Estruturas principais

B. USER

A seguinte informação é guardada para cada utilizador:

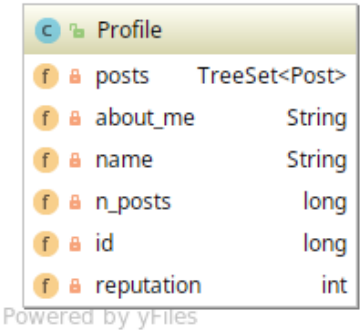

Figura 5. User

1) POSTS: Todos os *posts* do utilizador são guardados num *TreeSet* que segue a ordem natural dos mesmos (cronológica). Foi debatido guardar todos os *posts* ou apenas os últimos 10 e caso fosse esse o cenário se eram guardados num *BoundedTreeSet* ou numa *max heap*. Um *BoundedTreeSet* tem a vantagem de ser de relativamente fácil implementação mas a ordenação é mais custosa que uma *max heap*. No fim decidimos guardar todos os *posts* do utilizador visto terem utilidade para outras *queries* nomeadamente a *11*.

2) COMPARABLE: Foi discutido implementar uma ordem natural para os *users* mas visto que apenas é necessária em duas *queries* e por critérios diferentes (reputação e número de posts), não foi feito.

C. POST, QUESTION, ANSWER

1) POST: Como classe *mae* temos a *Post* que guardas as informações comuns tanto a questões como respostas. Esta classe implementa a *interface comparable* forçando a ordem natural ser cronológica (critério mais comum encontrado por nós).

2) ANSWER: As respostas apenas acrescentam um campo do tipo *long* que corresponde ao *id* da questão a que se refere.

3) QUESTION: Com o objectivo de inter-ligar as respostas com as questões decidiu-se guardar num *Map* todas as respostas relativas a uma dada questão onde a chave é o *id* respectivo.

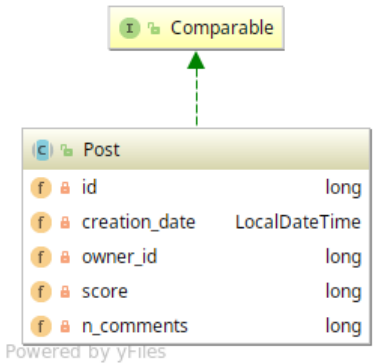

Figura 6. Post

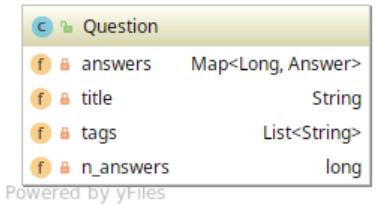

Figura 7. Question

D. TARDIS

A *TARDIS* foi a estrutura que mais alterações sofreu. Enquanto que na implementação em *C* consistia em duas "variaveis de classe", decidiu-se implementar apenas uma do tipo genérico *Post*. Esta mudança deu-se ao facto de não se ter ganho quase nada com a separação de perguntas e respostas. Outra mudança significativa foram as estruturas usadas para criar a *TARDIS* em si Enquanto que no projecto em *C* foram usados *Arrays* de *Arrays* para *GSequences* nesta implementação foram usados *HashMap* para *HashMap* para *List<Post>*. Cada chave corresponde a um indice (ou de anos ou de meses/dias).

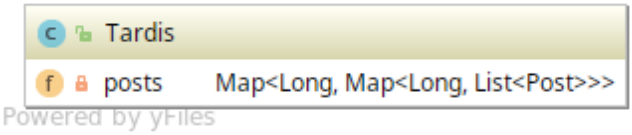

Figura 8. Tardis

1) ANAMORFISMO: De forma a obter *posts* dentro de um intrevalo de tempo e ordenados por um critério *ad hoc* foi implementado o metodo *getBetweenBy*:

Em primeiro lugar é interessante notar a assinatura do metódo *TreeSet<? extends Post>*. Restringimos o tipo de retorno com uma *bounded wildcard* sendo assim possível retornar tanto um *TreeSet* de *Post*, *Answer* ou Question. Os seus parâmetros também são bastante interessantes. De forma

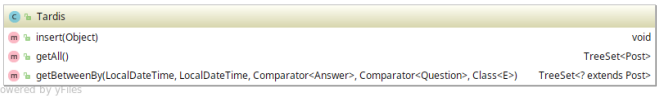

Figura 9. Metodos da Tardis

a saber que tipo procurar (ou tipos), passamos a Class de procura como um genérico Class<E> type onde o *genérico E* é *restringido* na assinatura do método <*E extends Post*>. Dado a falta de *function pointers* (tanto quanto sabemos) passamos um *Comparator* para ordenar o *TreeSet*.

```
List<Post> tmp = this.posts.entrySet()
                           .stream()
                           .filter(f -> f.getKey() >= index_ano_inicio && f.getKey() <= index_ano_fim)
                           .map(Map.Entry::getValue)
                           .flatMap(f -> f.entrySet()
                                          .stream()
                                          .filter(sf -> sf.getKey() >= index_mes_inicio
                                                        && sf.getKey() <= index_mes_fim)
                                          .map(Map.Entry::getValue)
                                          .flatMap(Collection::stream)

                           ).collect(Collectors.toList());
```

Figura 10. Stream da vida

Este método tem como gene uma stream. Isto torna o raciocinio muito mais simples do que com *forEach* ou iteradores externos. De facto, este método é um anamorfismo de *TreeSets*.

E. COMPARATORS EXCEPÇÕES

Entre *comparadores anonimos* foram ainda criados:

- PostCreationDateComparator –> Compara dois *posts* em relação à sua data de criação, de forma **descendente**;
- QuestionCreationDateComparator –> Compara duas *questions* em relação à sua data de criação, de forma **descendente**[1];
- ProfileNPostsComparator –> Compara dois users em relação ao número de posts, de forma **descendente**;
- AnswserScoreComparator –> Compara duas *answers* em relação ao seu score, de forma **descendente**;
- QuestionAnswerCountComparator –> Compara duas questions em relação ao número de respostas, de forma **descendente**.

Criou-se também as seguintes excepções:

- NoPostFoundException –> Caso um ID nao exista entre os Posts;
- NoProfileFoundException –> Caso um ID nao exista entre os Perfis;
- NoTagFoundException –> Caso um ID nao exista entre as Tags;
- PostIsNotOfRightTypeException –> Caso uma query exiga algo sobre um tipo de Post, mas recebe outro.

V. ESTRUTURAÇÃO DE QUERYS

A. QUERY1

Nesta *query* começamos por verificar o tipo do *Post* pedido. Caso seja uma *Answer* vai buscar a informação à *Question*

[1]Apesar de já existir um comparador igual para Posts, este foi necessário para o uso da Tardis, em alguns métodos.

a que pertence, caso contrário vai diretamente a esta. É importante referir que alteramos a interface **TADCommunity**, colocando na assinatura desta query a possibilidade de produzir **excepções** caso: não haja o Post ou o utilizador a que pertence, a Question não existe.

B. QUERY2

Todos os perfis existentes na estrutura principal num *TreeSet*, usando o comparador *ProfileNPostsComparator* que ordena perfis consoante o nº de posts deste (de forma descendente). De seguida, retorna apenas os primeiros *N* perfis, sob a forma de uma lista de longs (usando streams, maps e limites).

C. QUERY3

Como a nossa estrutura *Tardis* possui um método que retorna todos os Posts existentes no programa, apenas chamamos esse método, retornando um par apenas com o tamanho de cada tipo das *questions* e *answers*.

D. QUERY4

Nesta *query* começamos por recolher todas as *questions* entre as datas pedidas, através da *Tardis* (ordenadas por data). Depois retornamos uma lista com o ID de todas as *questions*. É importante referir que experimentamos a implementação com iteradores externos e internos, estando os resultados mais à frente no relatório.

E. QUERY5

Aqui acedemos ao *HashMap* da estrutura principal para receber o *Profile* com o *ID* passado. Depois retornamos um par com a *bio* deste *Profile* e os 10 posts mais recentes(através do TreeSet<Post> existente no Profile, apenas pegando os 10 primeiros). Fizemos também a experiência de ir buscar os 10 últimos *posts* deste (sendo os 10 primeiros no TreeSet do Profile) com iteradores internos e externos.

F. QUERY6

Usando a *Tardis* para ter todas as answers entre as datas pretendidas, ordenadas por um comparator *AnswerScoreComparator* que ordena por score (votos - descendente). Depois usamos um iterador externo para percorrer apenas as primeiras *N answers* ordenadas.

G. QUERY7

Começamos por recolher todas as *questions* compreendidas entre as datas pedidas, onde depois, percorremos cada uma contando as respostas, adicionando a um par constituído pelo número de respostas calculado e o *ID* da *question*. Adicionamos estes pares a uma *TreeSet* que ordena os pares por ordem decrescente da sua primeira componente (número de respostas), retornando assim a segunda componente do primeiro par do *Set*.

H. QUERY8

Percorrendo todas as *question* (que possuem título), recorrendo à *Tardis*, adicionamos as primeiras N Question ,o ID onde a palavra existe no título(usando iteradores externos).

I. QUERY9

Nesta *query* pegamos em todos os *posts* do primeiro utilizador passado e fazemos o seguinte *flow*:

- Se o *Post* for uma *Answer*, verificamos se a *Question* a que pertence a resposta pertence ao segundo utiizador. Se sim, inserimos num *TreeSet* ordenado por data *PostCreationDateComparator* caso ainda não exista neste;
- Se for uma *Question*, percorremos todas as respostas, verificando se alguma pertence ao segundo utilizador. Se sim, inserimos no *TreeSet* caso ainda nao exista.

Importante referir que: não vale a pena continuar a percorrer as respostas ou questoes quando já se encontrou algo em comum.

J. QUERY10

Nesta começamos por ir buscar o *Post* a que pertence o *ID*. Se não for uma *Question* atira uma excepção *PostIsNotOfRightType* , indicando que não se pode obter as respostas de uma reposta. Depois percorremos cada resposta, calculando o score desta e atribuindo a um par constituído por este e pelo ID da resposta. Caso algum score seja superior ao existente no par, atualiza-se este, retornando no fim a parte do par com o *ID* da reposta. Para facilitar, criamos um método que calcula o score de uma Answer.

K. QUERY11

Seleciona-se os top *N* utilizadores, colhe-se os seus *posts*, filtra-os pelas datas passadas como argumentos e guarda todas as tags presentes juntamente com o numero de vezes que aparecem. Finalmente limita-os por *N* e retorna uma lista de ids do mesmos. Tudo isto feito numa stream.

L. OPTIMIZAÇÕES

Durante a fase em que realizamos as *queries* testamos várias formas de as resolver: *Iteradores externos*, *forEach* e Streams. Na maioria das *queries* onde utilizamos as *streams* é necessário percorrer a *collection* toda, logo usar *iteradores* externos não oferece vantagens (valores de tempos oscilavam entre 10 a 50 ms)

M. BENCHMARKS

[2]

Comparativamente a performance em do projecto em *C* a *performance* do *parser* é pior. Por sua vez as *queries* tiveram um aumento de cerca de *5 a 10 ms* em geral. Este resultado ja era esperado dado *Java* ser traduzido para *bytecode* e não para instruções do CPU. A pior *query*, à parte do *load*, demora cerca de *10 ms*.

[2]Infelizmente devido as configurações do documento latex nao foi possivel criar uma tabela com os tempos.

1) Tempos java com streams:

- Load : 21858 ms;
- Query 1: 3 ms;
- Query 2: 270 ms;
- Query 3: 40 ms;
- Query 4: 12 ms;
- Query 5: 8 ms;
- Query 6: 5 ms;
- Query 7: 8 ms;
- Query 8: 202 ms;
- Query 9: 3 ms;
- Query 10: 13 ms;
- Query 11: 270 ms.

2) Tempos java com iteradores:

- Load : 21820 ms;
- Query 1: 3 ms;
- Query 2: 273 ms;
- Query 3: 20 ms;
- Query 4: 12 ms;
- Query 5: 6 ms;
- Query 6: 5 ms;
- Query 7: 3 ms;
- Query 8: 179 ms;
- Query 9: 3 ms;
- Query 10: 13 ms;
- Query 11: 180 ms.

A diferença é negligenciável.

3) Em C:

- Load : 17200 ms;
- Query 1: 0 ms;
- Query 2: 6 ms;
- Query 3: 5 ms;
- Query 4: 7 ms;
- Query 5: 10 ms;
- Query 6: 2 ms;
- Query 7: 1 ms;
- Query 8: 20 ms;
- Query 9: 0 ms;
- Query 10: 3 ms;
- Query 11: 20 ms.

VI. CONCLUSÃO

Esta segunda fase do projeto deu para verificar que o processamento em Java é mais lento relativamente ao trabalho em C. Por outro lado as Java Collections oferecem grande flexibilidade e uma maior liberdade. De facto, graças a elas deu para explorar mais implementações.

Em comparação à primeira fase, esta foi mais fácil. Não tivemos que criar qualquer estrutura comum ,visto que o Java contém muitas estruturas pré definidas com optimizações. Também não houve necessidade de libertar memória alocada, visto que o garbage collector faz isso por nós.

Uma das grandes vantages de Java é que oferece uma interface funcional o que permite estruturar o pensamento de outra forma, infelizmente em maior parte dos casos torna o código mais lento.