Relatório do Trabalho Prático de LI3

Grupo 42

João Queirós (A82422)

José Costa (A82136)

Luís Alves (A80165)

Miguel Carvalho (A81909)

12 de Junho de 2018

Conteúdo

1 Introdução

1.1 Contextualização

Este projeto foi desenvolvido no âmbito da Unidade Curricular de *Laboratórios de Informática III*, no qual analisaremos e processaremos informação do ***Stack Overflow***, ***Android*** e ***Ask Ubuntu***, recorrendo à linguagem de programação Java.

1.2 Objetivos

Após realizarmos o trabalho em C, propusemo-nos a melhorar a modularização deste, tal como reduzir ao mínimo as interdependências entre classes.

Ainda assim, tivemos sempre em mente a obtenção de reduzidos tempos de resposta para cada *query*.

2 Estrutura de Dados

De forma a organizarmos as nossas classes pelos packages previamente criados, optamos por colocar os tipos básicos de dados no package `li3`, assim como as interfaces que definiam métodos a implementar pela classe que armazenasse os dados da aplicação (`Community`).

Criamos um package extra `exceptions` onde colocamos, naturalmente, algumas exceptions que criamos para o projeto.

No package `common`, colocamos todos os *Comparators* usados nas diversas classes, enquanto que no package `engine` colocamos a classe de parsing (`Parser`), as classes respetivas a cada *query*, bem como a classe que implementa a `TADCommunity` (`ForumsModel`) e a classe que funciona como base de dados da aplicação (`Community`).

2.1 Tipos Básicos

2.1.1 Post

Um `Post` é um objeto simples que contém informações comuns a `Question` e a `Answer`: título, id, data de criação e id do criador. Este objeto é a superclasse das classes acima referidas.

2.1.2 Question

Uma `Question` é subclasse de `Post`, e contém uma lista com os nomes das tags dessa questão bem como o atributo *AnswerCount*.

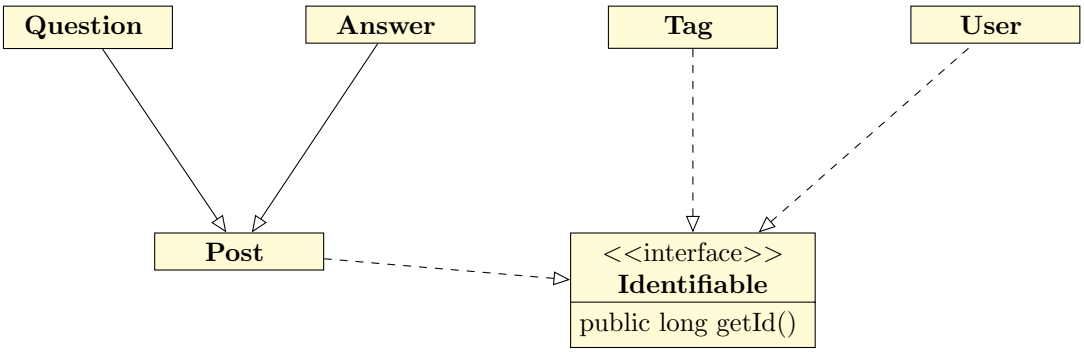

Figura 1: Diagrama dos Tipos Básicos

2.1.3 Answer

Uma `Answer` é subclasse de `Post`, e contém várias informações adicionais relativamente a este: id da questão que responde, o seu score e o nº de comentários.

2.1.4 Tag

Uma `Tag` é um objeto que define os tipos de tags que uma questão pode ter, por isso contém: o nome da tag e o id associado.

2.1.5 User

Um `User` é um objeto que representa um utilizador no sistema, e contém as seguintes informações: uma pequena bio, nome, id, reputação e nº de posts.

2.2 Armazenamento de Dados

De forma a posteriormente podermos consultar os dados de forma eficiente, optou-se por não criar clones dos objetos indiscriminadamente, já que estes, após serem carregados, não seriam mais alterados, já que todas as queries são meramente de consulta. Isto permitiu uma poupança significativa na memória ocupada, bem como no trabalho extra que seria necessário por parte do CPU para clonar todos os objetos. Ainda assim, consideramos que os dados ficam devidamente encapsulados, já que apenas permitimos o acesso via métodos públicos com um comportamento bem definido (as variáveis de instância permanecem privadas).

Assim sendo, criamos uma classe `Community` que agrega os dados da seguinte forma:

- Map de id para `User`
- Map de id para `Post`
- Map de id de uma `Question` para um Set de `Answer` (ordenado por ordem cronológica inversa)

- Map de id de um `User` para um Set de `Post` (ordenado por ordem cronológica inversa)
- Lista de `User` ordenado por ordem decrescente de nº de `Post`
- TreeMap de `LocalDateTime` para um Set de `Post`, ordenado por ordem cronológica inversa
- Map de nome de uma `Tag` para a respetiva `Tag`

Os Maps foram usados de forma a obter em tempo constante um determinado objeto/conjunto de objetos de acordo com o seu identificador. Quanto à lista, esta foi usada pois, aquando do momento de *Loading*, colocamos primeiro os utilizadores na `Community`. No entanto, não sabemos *à priori*, o seu número de posts. Por isso, esse número é apenas incrementado quando procedemos ao carregamento dos posts. Findo esse carregamento, podemos usar um *Comparator* na Lista, de forma a ordenarmos por número de posts.

Quanto ao TreeMap, este foi usado por disponibilizar o método `subMap` que se revelou muito eficiente na obtenção de posts entre determinados intervalos de tempo.

3 Carregamento de Dados

3.1 Parsing

Para efetuarmos o parsing dos ficheiros .xml, optamos por usar a *Streaming API for XML* (StAX), já que permitia não carregar o ficheiro .xml todo para memória, mas sim carregar "row a row", uma melhoria relativamente ao parsing feito em C.

Para isso, criamos uma classe `Parser` que apenas possui métodos de classe, para retirar informação útil dos ficheiros *Posts.xml*, *Users.xml* e *Tags.xml*, devolvendo listas com o conteúdo de acordo com os respetivos objetos (`Post`, `User` e `Tag`).

3.2 Loading

Ao efetuarmos o *loading* dos ficheiros, limitamo-nos a pegar nas listas resultantes dos métodos de classe da classe `Parser`, e a colocar cada um desses objetos no objeto `Community`. Ora esse objeto possui um conjunto de métodos (privados) que relacionam os posts, users e tags de acordo com as variáveis de instância definidas (ver secção 2.2).

O *loading* dos ficheiros é feito pela seguinte ordem: primeiro são armazenados os users, depois as tags e por fim os posts.

4 Modularização

De forma a ser possível desenvolver o código concorrentemente pelos 4 elementos do grupo, optamos por partir o problema o máximo possível para não haver interdependências ou, havendo-as, serem minimizadas, nomeadamente através da definição de interfaces que sinalizavam métodos que posteriormente

deveriam ser implementados para completar a funcionalidade de uma determinada *query*. Assim sendo, cada *query* também tem a sua própria classe com métodos de classe que trabalham sobre os dados disponibilizados pela `Community`.

Tendo isto em consideração, criamos os seguintes módulos fundamentais, que poderiam ser desenvolvidos simultaneamente:

- Parsing dos ficheiros XML
- Tipos básicos
- Definição da estrutura que albergasse os dados e métodos de disponibilização desses mesmos dados
- Queries

5 Queries

5.1 InfoFromPost (1)

Utilizando o método `getPost()` que recebe como argumento o id do `Post` que queremos consultar no objeto `Community` que contém os dados (daqui em diante todos os métodos chamados a um objeto não seja explicitado, assumem-se que sejam chamados ao `Community`).

Caso seja devolvido um `Post`, verifica-se se é uma `Answer` ou `Question`, pois se for a primeira, o título e nome devolvidos são os da questão e criador respetivos; se for a segunda, é devolvido o título e nome da resposta e do criador da resposta. Caso o `Post` não exista, ou não possua título/nome, é passado null como resultado ao `Pair`.

5.2 TopMostActive (2)

Utilizando o método `getUsersByNumberOfPosts()` que recebe como argumento o número de utilizadores, os `User` são transformados no seu respetivo id e é devolvida a lista com esses ids. O método primeiramente referenciado, limita-se a passar para uma nova lista de tamanho máximo N a lista ordenada de utilizadores de acordo com o número de posts. Refere-se novamente que, por não se alterar o objeto, se passa a referência ao objeto que está guardado na `Community`.

5.3 TotalPosts (3)

Nesta *query*, inicialmente havíamos pensado em implementar uma estratégia que minimizasse o uso de computações específicas à *query* dentro da classe *Community*. Assim sendo, optamos por obter a lista de posts dentro de um intervalo e, posteriormente, iterá-la e contar se eram instâncias de `Question`, ou de `Answer`. No entanto, isto implicava que percorrêssemos essa lista de posts mais que uma vez. Optamos por uma outra abordagem, que se enuncia de seguida.

Sendo o tempo de execução obtido demasiado elevado, optamos por, ao coletar os posts dentro de um intervalo na classe `Community`, contar de imediato se

eram uma `Question`, ou uma `Answer`. Verificou-se um aumento de performance na ordem das 4 vezes (análise mais detalhada na secção 6).

5.4 QuestionsWithTag (4)

Inicialmente, utilizamos o método `filterQuestionByInterval()` que recebe como argumentos as datas de início e de fim da *query* de forma a obtermos uma lista com `Question`s feitas nesse intervalo. De seguida, usando uma `stream`, filtramos as questões de forma a de seguida podermos ordenar as que contêm a tag dada por ordem cronológica inversa.

5.5 GetUserInfo (5)

Sendo esta *query* bastante simples, a nossa estratégia passou por obter a bio de um utilizador através do método `getBio()`, que dado um id, verifica se o utilizador existe na `Community` e, se existir devolve a sua bio. Caso não exista, ou não possua bio, devolve *null*.

Quanto aos 10 últimos posts, recorrendo ao método `get10LatestPosts()` que devolve os 10 posts mais recentes de um utlizador dado o seu id, obtemos uma lista com os respetivos posts, já que armazenamos para cada utilizador, um Set já ordenado por ordem cronológica inversa dos seus posts.

5.6 MostVotedAnswers (6)

Utilizando o método `filterAnswerByInterval()`, que recebe um intervalo de tempo e devolve todas as `Answers` feitas nesse período, apenas necessitamos de, recorrendo a uma `stream`, utilizar o método `sorted()` de acordo com um comparador de `Answers` por *score* e limitar o tamanho da stream a N. É por fim coletada para uma lista de ids das `Answers`.

5.7 MostAnsweredQuestions (7)

Inicialmente utilizamos o método `filterQuestionByInterval()` que, tal como os métodos com nome similar usados anteriormente, devolve uma lista com as questões realizadas no intervalo de tempo dado como argumento. De seguida, iteramos todas essas questões e colocamos num HashMap o id de cada questão como chave, e o número de respostas como valor. De seguida, fazemos `stream` do entrySet e usando o comparator `ComparatorLongIntEntryReverseInt` (que, dadas duas entries, devolve a com maior valor em primeiro lugar, e se os valores forem iguais, coloca a que tem maior key em primeiro lugar) ordena-se a stream limitando-a a N e coletando os ids das questões para uma lista.

5.8 ContainsWord (8)

Para esta *query*, utilizamos uma versão do método `filterQuestionByInterval()` que não recebe nenhum argumento e, por isso, retorna todas as questões presentes na `Community`. De seguida, utilizando uma `stream`, filtram-se as questões que contêm um título que não seja nulo e que contenha a palavra passada como argumento da *query*, recorrendo ao método `contains()` da classe `String`.

De seguida, é feita uma ordenação por ordem cronológica inversa limitada a N e são coletados os ids das questões para uma lista.

5.9 BothParticipated (9)

Para solucionarmos esta *query*, criamos dois sets: um para as questões em que participou o `User` 1, e outro para o `User` 2. As questões nas quais cada `User` participou são dadas pelo método `getQuestions()`, que dado um id, devolve um set (que pode ser vazio) das questões nas quais um utilizador interveio, quer tenha sido via `Question`, ou `Answer`.

De seguida, colocamos em `stream` o set de questões do `User` 1 e filtramos aqueles que também fazem parte do set de questões do `User` 2, através do método `contains()` da classe `Set`. De seguida, ordenam-se as questões por id e limita-se o tamanho a N, coletando os ids das questões numa lista.

5.10 BetterAnswer (10)

Começamos a solucionar esta *query* obtendo o set de respostas de uma dada pergunta através do método `getAnswers()`. A partir daí, para cada resposta é calculado o seu valor e adicionado a um TreeMap, que associa a cada *score*, um set de ids de `Answer`, ordenado por ordem decrescente.

Por fim, retira-se o valor à cabeça do TreeMap e é o id pretendido.

5.11 MostUsedBestRep (11)

Sendo esta a *query* mais complexa (e que mais dados necessita de cruzar), foi a que conseguimos um pior tempo de execução. O processo para obter o resultado é o que se enuncia a seguir.

Inicialmente, é obtido o Set de `Questions` que foram publicadas no intervalo dado. A partir desse Set, foi criada uma lista de `Users` num método auxiliar (`usersThatParticipated()`) que publicaram esses `Posts`, sendo depois essa Lista ordenada por reputação e truncada a N.

Tendo já a lista final de N utilizadores com melhor reputação, por cada utilizador nessa lista, obtemos o Set de posts desse `User`, sendo que caso o `Post` seja uma `Question` e o primeiro Set de `Posts` obtido contiver este `Post`, então é adicionado o id da `Tag` e o número de vezes que foi usada a um HashMap.

Por fim, as *entries* desse Map são colocadas em `stream` e ordenadas de acordo com os valores (e, caso os valores sejam iguais, por ordem crescente de ids), sendo coletados os ids para uma lista truncada a N.

6 Análise de Performance

Num computador portátil com 8GiB de RAM e um processador Intel® Core™ i7-3630QM @ 2.40GHz com 8 cores, obtivemos os mesmos resultados que os de referência (atualização 3), tendo registado os tempos na tabela abaixo.

Para além disso, notamos algumas diferenças quando implementamos diferentes abordagens nalgumas *queries*.

Na *query* 3, em vez de pedirmos os posts entre datas por parte da nossa TCD, e depois iterá-los e verificar se eram `Question` ou `Answer`, optamos por na própria `Community`, ao recolher os posts, devolver imediamente o par indicativo do número de perguntas e de respostas. Isto permitiu passar de 100ms, para 25, no parâmetro 2.

	Tempos	
Queries	Parâmetro 1	Parâmetro 2
1	0ms	1ms
2	7ms	7ms
3	7ms	25ms
4	28ms	25ms
5	1ms	1ms
6	14ms	7ms
7	7ms	68ms
8	148ms	120ms
9	3ms	2ms
10	1ms	1ms
11	25ms	145ms

Tabela 1: Tempos registados

Quanto às *queries* que necessitavam de posts dentro de um dado intervalo, inicialmente havíamos optado por guardar esses posts num Set ordenado por data. No entanto, obter um subconjunto não se revelou muito eficiente, e por isso optamos por usar um TreeMap, que graças ao método `submap` permitia obter o range desejado mais rapidamente. Após implementarmos esta mudança na `Community`, verificamos que o tempo de execução de cada *query* que requeria posts dentro de um intervalo diminui para cerca de 1/6 do tempo original com um Set.

Uma nota também para o facto de a *query* 8 ser a mais demorada, já que não organizamos o título de cada post de uma forma específica (dicionário, por exemplo) de forma a consultar se a palavra estava contida no título de forma ainda mais eficiente.

7 Conclusão

Após a conclusão deste projeto, verificamos que este ficou com muito menos interdependências que o anterior em C. Para além disso, durante o desenvolvimento, notamos que este podia ser feito mais concorrentemente sem tantos problemas de *merge*. Assim sendo, pensamos que cumprimos o planeado, tendo obtido bons tempos de execução em cada *query*, boa modularização e código simples de modificar e fazer manutenção.