Projeto em Java de Laboratórios de Informática 3
Grupo 5

Henrique José Carvalho Faria a82200 José André Martins Pereira a82880
Ricardo André Gomes Petronilho a81744

12 de Junho de 2018

Resumo

No contexto da disciplina de Laboratórios de Informática 3 do Mestrado Integrado em Engenharia Informática da Universidade do Minho foi nos proposto a elaboração de um projeto um projeto no paradigma de Programação Orientada a Objetos (POO) na linguagem Java. O projeto consiste na manipulação e tratamento de uma base de dados armazenada em ficheiros com formato - **.xml** - com o objetivo de responder a interrogações - **queries** - sobre a base de dados através de uma estrutura de dados.

Objetivos: foi atribuído como objetivos do grupo: obter eficiência nos tempos de execução de cada query e enrobustecimento do programa com **tratamento de excepções** e **encapsulamento** a nível de cada class.

Metodologia: no desenvolvimento do projeto foi seguido as convenções de POO. Foram definidos os métodos `get()` e `set()` com o formato - get`NomeDaVariavel`() e set`NomeDaVariavel`(). Foram criados também os métodos `clone()`, `toString()` e `equals()` paras as classes relevantes. Em todos os métodos responsáveis por entrada e saída de dados, o encapsulamento foi assegurado evocando o método clone(), excepto nos objetos do tipo String ou classes de Wrapper (Integer, Long, etc) uma vez que não é necessário. O uso de stream de dados foi evitado, excepto quando muito conveniente, uma vez que o processamento de um grande volume de dados numa stream é menos eficiente. Os métodos que necessitam de ordenar dados utilizam para tal um `Comparator<T>`. O projeto foi desenvolvido usando o ambiente de desenvolvimento integrado (IDE) - `IntelliJ`.

Resultados: os tempos de execução e resultados de cada query são possíveis de se observar nos ficheiros times.txt e results.txt respetivamente, quando compilado e executado o ficheiro run.sh.

Conclusão: Findo o projeto foram obtidos os resultados pretendidos. Note-se que o tempo de execução de cada query poderia ser menor caso a metodologia utilizada não se focasse no encapsulmento - a segurança e bons hábitos de programação abordados pesaram no tempo de resposta.

Palavras-chave: xml, query, Java, POO, classe, encapsulamento, excepção

Conteúdo

1 Classes

Na evolução do projeto foi necessária a criação e utilização de vários tipos de dados. De seguida estão listadas apenas as classes utilitárias, localizadas no package main.java.common.

Note-se que todas estão agrupadas numericamente uma vez que é possível distinguir nitidamente o motivo da sua existência no nosso programa. O grupo (1) destina-se a armazenamento de informação. O grupo (2) tem o propósito de realizar o parser dos dados contidos nos ficheiros .xml para as estruturas de armazenamento. Por último, o grupo (3) tem a finalidade de aumentar a segurança e robustez do nosso programa.

- Dados (1)
- Post (1)
- User (1)
- Tag (1)
- MyLog (1)
- Pair (1)
- PostsParser (2)
- UsersParser (2)
- TagsParser (2)
- IdNegativoException (3)
- PostExisteException (3)
- PostNaoExisteException (3)
- PostNaoEUmaPerguntaException (3)
- TagExisteException (3)
- TagNaoExisteException (3)
- UserExisteException (3)
- UserNaoExisteException (3)
- UserNaoTemPostsException (3)
- ValoresInvalidosException (3)

A seguir são apresentadas as classes que implementam as queries e funcionalidades relacionadas, localizadas no package main.java.engine.

Novamente, estão agregadas entre si. A agregação (4) refere-se á implementação direta das queries. O conjunto (5) concretiza o método de ordenação usado para a execução das queries. O aglomerado, neste caso unitário, (6) implementa a classe que acarreta a execução de todas as queries.

- Q1 (4)
- Q2 (4)
- Q3 (4)
- Q4 (4)
- Q5 (4)
- Q6 (4)
- Q7 (4)
- Q8 (4)
- Q9 (4)
- Q10 (4)
- Q11 (4)
- ComparatorPostAnswerCount (5)
- ComparatorPostData (5)
- ComparatorPostMaisVotos (5)
- ComparatorTagsMaisUsadas (5)
- ComparatorUserMaiorReputacao (5)
- ComparatorUserNumeroPosts (5)
- STDSort (5)
- TCDExample (6)

Desta vez é enumerada a definição da interface do sistema de queries e as classes de testes, localizadas no package main.java.li3.

O grupo (7) engloba todas as classes de testes. O conjunto (8) e (9), ambos unitários, definem a interface do sistema de queries e o método main, respetivamente.

- TesteQ1 (7)
- TesteQ2 (7)
- TesteQ3 (7)
- TesteQ4 (7)
- TesteQ5 (7)
- TesteQ6 (7)

- TesteQ7 (7)
- TesteQ8 (7)
- TesteQ9 (7)
- TesteQ10 (7)
- TesteQ11 (7)
- TesteTag (7)
- TesteUser (7)
- TestePost (7)
- TesteDados (7)
- TADCommunity (8)
- Main (9)

1.1 Class Tag

A class `Tag` representa a informação de uma tag, que tem as seguintes variáveis de instância:

```
private long id;
private String nome;
```

Cada `Tag` é representada pelo seu id e o nome. A razão para a criação deste tipo de dados, foi a resolução da query 11, onde se pertende as N tags mais usadas pelos N utilizadores com melhor reputação.

Esta classe destina-se apenas a servir como ligação entre o nome da Tag e o respetivo id.

1.2 Class Post

A class `Post` representa a informação de um post, que tem as seguintes variáveis de instância:

```
private long id;
private int tipo;
private long userId;
private long parentId;
private String titulo;
private String tags;
private LocalDate data;
private long answerCount;
private int score;
private int commentCount;
private int upvotes;
private int downvotes;
```

Para representar a data do post utilizou-se o tipo `LocalDate`, facilitando assim a manipulação das mesmas, através dos métodos `isEqual`, `isBefore`, `isAfter` entre outros. Das variáveis de instância representativas do `Post`, é importante realçar o significado de `tipo`, que informa se o `Post` é um pergunta (tipo = 1), respostas (tipo = 2) ou outro (tipo > 2). O `parentId` identifica, em caso do `Post` ser uma resposta, o id da pergunta responsável por essa resposta.

1.3 Class User

A class `User` representa a informação de um utilizador, que tem as seguintes variáveis de instância:

```
private long id;
private int reputacao;
private String nome;
private String bio;
private List<Long> postsIds;
private int upVotes;
private int downVotes;
private int numeroPosts;
```

Das variáveis acima, é importante realçar o significado dos `postsIds` que é um `List<Long>` onde cada `Long` corresponde a um id de um `Post` criado por este utilizador. Armazena-se apenas o id de cada `Post` deste utilizador, evitando-se informação repetida, uma vez que para obter a restante informação do `Post` procura-se o mesmo através do seu id contido num obejto da class `Dados`. Usou-se o tipo `List`, visto que se pretende uma estrutura de dados que possa crescer não sendo relevante a procura direta, mas simplesmente iterar os ids dos posts. O `numeroPosts` representa a soma dos posts que são perguntas e respostas, esta informação foi bastante útil na resolução da query 2.

1.4 Class Dados

A classe `Dados` tem como objetivo conter toda a informação resultante do parser dos ficheiros `.xml`. O propósito da criação desta class, com a respetiva composição, foi a necessidade da existência de um tipo de dados que contivesse toda a informação necessária para o processamento das queries. Assim a class Dados é composta pelas seguintes variáveis de instância:

```
private Map<Long, User> users;
private Map<Long, Post> posts;
private Map<String, Tag> tags;
```

As estruturas de dados utilizadas para guardar as diferentes informações foram do tipo Map, pois este permite a organização da informação associada a uma chave (`key`), que neste caso foram os ids respetivos do `User`, `Post` e `Tag`. Os tipos `User`, `Post` e `Tag`, contém toda a informação necessária para o processamento das queries, inclusive o id, sendo plausível considerar que é informação repetida ter a key - id - no respetivo tipo, mas do ponto de vista

de reutilização destas classes em futuros projetos, decidiu-se manter o id em cada um dos tipos.

A nível de **encapsulamento**, como foi referido anteriormente, seguiu-se a convenção de POO, garantido em todos métodos onde se verifica a entrada e saída de dados. De seguida apresenta-se um exemplo de um desses métodos, onde se assegurou encapsulamento, com a aplicação do método `clone()` que está definido nos tipos `User`, `Post` e `Tag`:

```
/**
 * Adiciona um User aos dados.
 *
 * @param user User
 */
public void add(User user) throws UserExisteException{
    if(this.users.containsKey(user.getId()))
        throw new UserExisteException(Long.toString(user.getId()));
    this.users.put(user.getId(), user.clone());
}
```

Note-se que todos os métodos que adicionam ou procuram informação na class Dados estão preparados para o **tratamento de exepções**, evitando assim que, no caso de algo correr mal na execução do programa, o mesmo não atinga um erro fatal. No método acima, na ocorrência de uma adição de um User já existente, é lançada um expção informando o id do User equivocado, sendo depois tratada na class responsável para o efeito.

Para além dos métodos convencionais get(), set(), add() entre outros, definiram-se outros métodos com igual importância como por exemplo o `getPostsPorDataCollection()`, que devolve uma coleção de posts criados num dado determinado intrevalo de tempo. Este método é extremamente útil nas queries que necessitam de informação dos posts dentro de um intervalo de tempo especificado. De seguida apresenta-se a implementação deste método, onde se garante também encapsulamento, visto que o método `getPosts()` já o faz:

```
/**
 * Devolve uma coleção de posts ordenado por data de criação
 * dentro do intervalo de tempo especificado.
 *
 * @param inicio data de inicio do intervalo de tempo
 * @param fim data do fim do intervalo de tempo
 * @return retorna uma coleção de posts ordenada
 */
public Collection<Post> getPostsPorDataCollection(LocalDate inicio, LocalDate fim){
    return this.getPosts().values().stream().filter(post -> !post.getData().isBefore(inicio) &&
            !post.getData().isAfter(fim))
        .sorted(new ComparatorPostData())
        .collect(Collectors.toCollection(ArrayList::new));
}
```

Como se pode observar, foi utilizado um comparator - `ComparatorPostData` - que ordena a stream de posts, sendo o critério de ordenação a data de criação do post. O retorno deste

método é bastante conveniente visto que, com uma `Collection<Post>`, podemos iterar os posts continuando a ser um tipo de dados - `Collection<T>` - bastante genérico e por isso facilmente reutilizável.

Do mesmo modo existe o método denominado `getUsersPorDataCollection()` que retorna uma coleção de users que criaram posts num dado intervalo de tempo.

Finalmente é de realçar o método `writeOnFile()`, responsável por escrever todos os dados num ficheiro especificado em formato de texto. Assim, foi útil no desenvolvimento da class Dados, uma vez que permitiu ao grupo realizar um **debugging** visual para verificar correção do parser e mesmo de algumas queries.

2 Modularização funcional

De forma a facilitar o desenvolvimento do projeto e visto que a linguagem Java tem a modularização como convenção, separou-se o projeto em módulos/class diferentes . Assim permitiu-se uma maior organização entre o grupo de trabalho, de forma autónoma sem haver conflitos, em partes diferentes do mesmo. A reutilização do código foi simplificada e facilitada.

Na implementação do código, inclui-se uma API pública e privada, este procedimento permite uma coperação autónoma uma vez que é possível ao longo do desenvolvimento de cada módulo/class a alteração da implementação de cada função, e no entanto é presevada a sua API.

3 Abstração de dados

A abrastação de dados foi garantida, em todas as classs, tornando o código mais robusto, pois cada elemento do grupo, não necessitou obrigatoriamente de saber a implementação interna, para poder utilizar a respetiva API.

Do mesmo modo, se garantiu o encapsulamento dos dados, ou seja, todas as entradas e saídas de dados, são feitas com cópias e não com os valores originais, sendo uma escolha do grupo, garantir o mesmo, apesar de ser mais custoso a nível de tempo de resposta, mas por outro lado, garante-se segurança dos dados, que se achou mais importante.

4 Estratégias seguidas em cada uma das interrogações

4.1 Interrogação 1

O primeiro passo foi verificar se o id dado como argumento corresponde ao id de uma pergunta ou de uma resposta. Caso seja uma pergunta basta obter o nome e título do utilizador que a criou. Tratando-se de uma resposta identifica-se a pergunta correspondente e de seguida procura-se o nome de utilizador da mesma.

4.2 Interrogação 2

Inicialmente copia-se todos os users para uma Lista, para que se possa ordenar os users pelo seu número de posts (perguntas e respostas), através do `ComparatorUserNumeroPosts`. Por fim, da lista ordenada, copia-se os N ids destes users para outra lista.

4.3 Interrogação 3

Dado um intervalo de tempo arbitráio, percorre-se os posts compreendidos entre este intrevalo, verificando-se o tipo de cada um, incrementando o contador respetivo, caso seja uma pergunta ou uma resposta. No fim é retornado o número de perguntas e respostas.

4.4 Interrogação 4

Percorrem-se os posts que estão no intrevalo usando o método `getPostsPorDataCollection`, filtrando os que são perguntas e que contém a tag, sendo que foi bastante útil o método `contains`, da class String, que verifica se uma String está contida em outra. Após esta filtragem, procede-se à colocação dos ids dos posts por cronologia inversa, usando o método `sortPostsByInvertedCronology` da class `STDSort`.

4.5 Interrogação 5

Dado o id do utilizador, obtêm-se a biografia - `shot_bio` - do mesmo e 10 últimos posts - `post_history` - criados pelo utilizador, tomando partido do método `sortPostsByInvertedCronology()`, para ordenar o posts por ordem decrescente da sua criação. Note-se que caso que o número de posts seja inferior ao N dado, apenas se retorna uma lista contendo esse número de posts.

4.6 Interrogação 6

Percorrem-se os posts que estão no intervalo usando o método `getPostsPorDataCollection`, e através do `ComparatorPostMaisVotos` obtêm-se por ordem de número de votos decrescente os posts com mais votos. De seguida adicionam-se à lista a devolver os posts que são respostas.

4.7 Interrogação 7

Percorrem-se os posts que estão no intrevalo usando o método `getPostsPorDataCollection`, de seguida usa-se um for para separar para uma lista auxiliar apenas os posts que são perguntas, sendo depois estas ordenadas por ordem decrescente de respostas com recurso ao `ComparatorPostAnswerCount`. Por fim colecionam-se numa lista as N perguntas com mais respostas ou todas as perguntas caso estas estejam em menor número que N.

4.8 Interrogação 8

Dado uma tag, percorre-se uma coleção contendo todos os posts, verificando-se se o post é um pergunta e se a tag dada está contida no título do post, em caso afirmativo acrescenta-se o id do respetivo post ao Arraylist que armazena os ids dos posts. No final tomando partido do método `sortPostsByInvertedCronology`, ordena-se o Arraylist dos posts por ordem decrescente da sua data de criação. Note-se que caso que o número de posts seja inferior ao N dado, apenas se retorna uma lista contendo esse número de posts.

4.9 Interrogação 9

Compara-se os posts de cada utilizador e verifica-se os diferentes casos possíveis:

- o post do user 1 é pergunta e o post do user 2 é uma resposta a esta pergunta e caso contrário;
- o post do user 1 e user 2 são respostas à mesma pergunta;

Os posts que verificavam estes casos, os seus ids são adicionados a uma Lista, que por sua vez vai ser ordenada por cronologia inversa, usando o método `sortPostsByInvertedCronology` da class `STDSort`.

4.10 Interrogação 10

Dado o id de uma pergunta percorre-se uma coleção contendo todos os posts procurando-se por respostas à mesma. Caso uma resposta seja encontrada, identifica-se o utilizador que a cirou e obtem-se a reputação e votação desse utilizador, obtem-se também o score e número de comentários da resposta. Aplica-se estes quatro valores obtidos a uma função dada que calcula a respetiva média. Utiliza-se este procedimento para todas as respostas encontradas, guardando-se a maior média e o respetivo id dessa resposta.

4.11 Interrogação 11

Percorrem-se os users que fizeram perguntas no intrevalo usando o método `getUsersPorDataCollection`, sendo depois selecionados os N melhores ou todos caso estejam em menor número que N. De seguida colecionam-se os posts de todos esse utilizadores e filtram-se aqueles que não são perguntas ou que não foram feitos no intervalo de tempo dado. Daqueles que passam na seleção retiram-se as tags e separam-se em pares de (`nome` da tag,contador de ocorrências do mesmo nome no resto da lista). De seguida procede - se à contagem das tags na lista de pares e recorre-se ao `ComparatorTagsMaisUsadas` para ordenar as tags por ordem decrescente de ocorrência e em caso de oempate de corrência desempata-se usando o `id` da tag sendo depois as N mais usadas colecionadas e devolvidas.

5 Conclusão

Concluido o projeto, os objetivos incialmente propostos foram correspondidos. No final da elaboração do trabalho verifica-se a **eficiência e facilidade na proramação que a linguagem Java nos cedeu**, sendo mais intuitiva, facilitada para trabalho em equipa, organizada e com imensas classes e estruturas de dados pré-definidas que foram bastante úteis.

O **tratamento de exceções** foi importante para que o programa não tenha um erro fatal quando algo corre inesperadamente.

O controlo de memória que Java oferece - **Garbge Collector** - foi imensamente útil para que nós, programadores, não tenhamos preocupações com gestão de memória, sendo o esforço mais focado no código mais relevante, visto que no mesmo projeto elaborado em C, uma das principais dificuldades foi a gestão de memória, para que não houvesse **memory leaks**.

Note-se que o resultado de algumas queries não foram exatamente iguais aos resultados de referência fornecidos pelos professores, visto que, o critério de comparação de posts no mesmo dia ser diferente, colocando-os por uma ordem diferenciada dos resultados de referência. No entanto, quando tal aconteceu, verificamos manualmente que as datas eram idênticas para nos certificarmos que a implementação está correta.

Por fim refletiu-se que este projeto foi bastante enriquecedor a todos os níveis para o grupo, pois ensionou-nos a trabalhar em equipa, dividir tarefas, saber procurar por código já existente, valorizando a reutilização de código proveniente da linguagem Java e convenção de POO.