Primo progetto intermedio A.A. 2020-2021

Nicola Vetrini, Matricola 600199

Il progetto consiste nella realizzazione di componenti software per la gestione e l'analisi di una rete sociale (MicroBlog) basata su post testuali. Ho diviso l'implementazione in una classe Post ed una classe SocialNetwork, che ha il compito di memorizzare, analizzare e modificare la rete sociale creata dai post. Inoltre ho definito le seguenti eccezioni (nei rispettivi file)

- **SelfLikeException** sollevata se si tenta di inserire l'autore del post tra gli utenti che hanno messo like
- **TextOverflowException** sollevata se il testo del post supera i 140 caratteri
- **DuplicatePostException** sollevata se si tenta di aggiungere un post alla rete sociale, ma esiste già un post con tale id
- **NoSuchPostException** sollevata se si tenta di rimuovere un post da un'istanza di SocialNetwork, ma tale post non è presente

Sono tutte sono sottoclassi di **Exception**, pertanto sono checked, in modo da ridurre la possibilità che nel codice che usa i tipi di dato definiti si presentino errori a runtime. Nel codice ho usato anche l'eccezione **NullPointerException** (unchecked, predefinita da Java) per controllare che gli argomenti passati ai metodi non siano null, dato che in tal caso l'unica possibilità è non eseguire alcuna azione e ritornare al chiamante un errore (tranne per il metodo equals).

La classe Post

Post è un tipo di dato astratto modificabile costituito da una quintupla:
(id, autore, testo, timestamp, likes). I like li ho implementati tramite un Set<String> in modo da garantire che un utente possa comparire al più una volta.

Funzione di astrazione e invariante di rappresentazione

La funzione di astrazione è banale: le variabili d'istanza corrispondono ciascuna ad uno dei campi della quintupla che definisce un elemento di Post.
Per quanto riguarda invece l'invariante di rappresentazione, oltre alle usuali condizioni che nessuna variabile sia null e this != null, impongo che nessun elemento del Set di like sia nullo e che sia diverso dall'autore del post. Inoltre, essendo un insieme, non devono esservi duplicati (mi sembra ragionevole imporre che un utente possa mettere al più un like ad un post). Come specificato dal testo ho anche imposto che i post non possano avere più di 140 caratteri.
La rappresentazione concreta della classe ha quindi cinque variabili d'istanza (private) e una statica (sempre privata):

- Un identificativo univoco, che ho modellato come un **Integer**
- L'autore del post, che ho rappresentato tramite una **String**
- Il testo del post, una **String** lunga al massimo 140 caratteri
- La data ed ora di pubblicazione del post, sotto forma di una variabile di tipo **Date**
- Un **Set<String>** di like che altri utenti della rete hanno messo al post

La variabile statica di tipo **Random** serve a generare gli id quando il costruttore sottostante è chiamato.

```
public Post(String author, String text, Date timestamp) throws
TextOverflowException, NullPointerException
```

Solo l'insieme di like è modificabile, mentre le altre componenti dello stato possono essere soltanto lette una volta istanziate, tuttavia il tipo di dato globalmente è modificabile.
Ho usato il tipo **Date** per il timestamp poiché consente di stampare data ed ora, come richiesto; esiste anche un tipo di dato **Timestamp**, ma dal momento che l'unico

utilizzo che faccio della variabile è la stampa non ho ritenuto necessario usarlo, anche se potrebbe essere un'alternativa valida.
Il costruttore sopra riportato non è l'unico presente. Ne ho aggiunto un secondo che prende anche un parametro id di tipo Integer: il motivo di questa aggiunta è agevolare il testing: utilizzando tale costruttore posso creare volutamente conflitti tra gli id dei post e verificare che venga sollevata l'eccezione **DuplicatePostException** da parte dell'istanza di **SocialNetwork** alla quale ho tentato di aggiungere il post in questione. Infatti, dato che gli id sono generati (pseudo)casualmente l'evento di generare un id già presente ha probabilità molto bassa su un limitato numero di post, anche se non è da sottovalutare in un contesto reale.
L'insieme di like viene inizializzato all'insieme vuoto perché è presente un metodo aggiuntivo che permette di aggiungere like al post all'interno della classe, illustrato nel seguito.
Ho scelto un **HashSet<String>** per istanziare **this.likes** (di tipo **Set<String>**) in quanto l'HashSet garantisce una buona efficienza se confrontato con altre implementazioni dell'interfaccia Set e dato che non è necessario sfruttare l'ordinamento relativo tra gli elementi del Set.

Metodi aggiuntivi

Ho aggiunto, oltre ai metodi che ritornano copie dei valori dello stato interno, i seguenti metodi

- `public void addLike(String follower) throws NullPointerException, SelfLikeException`
- `public Set<String> getLikes()`
- `public String toString()`
- `public boolean equals(Post other)`

Il metodo **addLike()** aggiunge, dopo aver controllato che non sia null e diverso dall'autore del post, la stringa passata come parametro a this.likes se non era già presente. Se **follower == null** solleva **NullPointerException**, mentre se **follower.equals(this.author)** viene sollevata l'eccezione **SelfLikeException** (ed il like non viene aggiunto).
Il metodo getLikes ritorna semplicemente un nuovo oggetto di tipo Set<String> che contiene una copia dei valori in **this.likes**. Siccome le String sono un tipo di dato immutable, anche se il Set ritornato dovesse essere modificato, lo stato interno della corrispondente istanza di Post rimarrebbe invariato.
Il metodo **toString()** che ho sovrascritto è utile per ritornare la rappresentazione del Post come String nel formato specificato dalla funzione di astrazione, ovvero una quintupla. Il testo del post è tagliato ai primi venti caratteri per migliorare la leggibilità dell'output.
Il metodo **equals()** che ho sovrascritto prende come parametro un altro Post. Se **other == null** ritorna false, altrimenti confronta i campi id e ritorna true se e solo se l'id di this e quello di other sono uguali, senza confrontare le altre variabili d'istanza, in quanto gli id dei post devono essere univoci. A questo proposito si potrebbe fare la considerazione che potrebbe essere consentita l'esistenza di post in istanze diverse di **SocialNetwork** che abbiano id uguale. Questo non avviene nella mia implementazione, ma d'altra parte non è molto significativo considerare il caso di post fuori da una rete sociale ed inoltre l'unicità dei post ha rilevanza (ed è infatti controllata) all'interno di una stessa istanza di **SocialNetwork**.

La classe SocialNetwork

La classe SocialNetwork è un tipo di dato astratto modificabile composto da una tripla: due funzioni (mappe) ed una lista di post. La specifica richiedeva soltanto che fosse presente una Map<String, Set<String>>:
user → {utenti seguiti da user}
ma ho ritenuto utile aggiungere anche la mappa
user → {utenti che seguono user}

Tali mappe le ho chiamate rispettivamente **following** e **followers**. Per rappresentare il follow ho adottato la convenzione che A segue B ⇔ A ha messo like ad almeno un post di B (presente nella rete sociale). La lista **posts** contiene tutti i post presenti nella rete, allora ho il seguente elemento tipico:
(followers: K → V, following: K → V, posts) dove posts = [post_1, …, post_n]
Ho implementato la possibilità di aggiungere dei post alla rete e di eliminarli da essa tramite funzioni apposite che aggiornano anche le mappe. Non è possibile modificare un post (aggiungere like ad esempio) tramite questa classe, in quanto ho ritenuto quella funzionalità più attinente ad un'altra classe (ad esempio una classe User, non implementata, che ottenga i Post nella rete con getPosts() e poi, in base ad una qualche azione esterna, aggiunga like tramite il metodo apposito della classe Post).

Funzione di astrazione e invariante di rappresentazione

La funzione di astrazione è banale: le variabili d'istanza corrispondono ciascuna ad uno dei campi della tripla che definisce un elemento di SocialNetwork.
Per quanto riguarda invece l'invariante di rappresentazione, oltre alle usuali condizioni che nessuna variabile sia null e che this != null si aggiungono i requisiti che ogni post di **this.posts** deve rispettare l'invariante di rappresentazione di Post e che entrambe le mappe rispecchino la rete sociale che si viene a creare considerando la lista di post, ovvero che in ogni possibile stato risulti che tutti i follower dell'autore sono mappati in **this.followers** e tutti i following in **this.following** (una formulazione più rigorosa è presente nel file).

Metodi aggiuntivi

In aggiunta ai metodi richiesti ho implementato anche i seguenti:

- `private void updateFollowers(Post post, Map<String, Set<String>> followAuth) throws NullPointerException`
- `private void updateFollowing(Post post, Map<String, Set<String>> iFollow) throws NullPointerException`
- `private void rmFromMaps(Post p) throws NullPointerException`
- `public void addPost(Post p) throws NullPointerException, DuplicatePostException`
- `public void rmPost(Integer pid) throws NullPointerException, NoSuchPostException`
- `public Map<String, Set<String>> guessFollowing(List<Post> ps) throws NullPointerException`
- `public List<String> influencers()`
- `public List<Post> getPosts()`
- `public boolean equals(SocialNetwork other)`
- `public String toString()`

I primi due metodi sono chiamati internamente per aggiornare le mappe passate come parametro quando un post viene aggiunto: **updateFollowers()** aggiorna il Set di utenti che seguono l'autore di post unendo a quello corrente il Set di like del post (oppure aggiungendo l'associazione {user} → {post.getLikes()} se la chiave non era presente). Duale il comportamento di **updateFollowing()** che unisce il set di utenti seguiti da chi ha messo like al post con {post.getAuthor()}, oppure aggiunge alla mappa la coppia chiave-valore {user} → {post.getAuthor()} se la chiave non era presente.
rmFromMaps() invece si occupa della situazione opposta: quando un post viene eliminato devo eliminarlo dalla lista di post e aggiornare entrambe le mappe. Eseguo prima l'operazione di rimozione dalla lista per semplificare l'aggiornamento delle mappe, che a questo punto non deve curarsi di escludere il post rimosso. Devo togliere dai seguiti di tutti gli utenti l'autore del post rimosso se e solo se essi avevano messo like a tale autore soltanto nel post rimosso. In tal caso devo inoltre togliere

l’utente dalla mappa di followers dell’autore del post. Chiaramente, se l’autore non ha più alcun post nella rete, i suoi followers devono essere un Set vuoto.
addPost() e **rmPost()** si limitano ad aggiungere/rimuovere i post dalle liste e poi chiamare le funzioni descritte sopra. La prima ha come argomento un post e se ne trova uno in **this.posts** con lo stesso id solleva **DuplicatePostException**, assicurando l’unicità del post all’interno dell’istanza di **SocialNetwork**. La seconda prende un id (Integer) e lo cerca nella lista di post: se lo trova elimina il post corrispondente, altrimenti solleva **NoSuchPostException**. La ragione di questa differenza sta nel fatto che per identificare un Post da rimuovere è sufficiente possedere il suo id, mentre per inserirlo devo necessariamente passare l’oggetto in questione come argomento.
guessFollowing(), in contrapposizione a **guessFollowers(),** che ritorna la mappa
{utente} → {utenti seguiti dall’utente chiave},
ritorna invece la mappa
{utente} → {utenti che seguono l’utente chiave}.
L’ho aggiunta principalmente per implementare facilmente **equals()**, dato che in esso devo verificare che le reti sociali coincidano esattamente, ma è public perché è comunque una funzione molto utile per ottenere una visione d’insieme della rete sociale.
Il metodo **influencers()** (senza parametro) restituisce la lista composta da tutti gli utenti della rete che hanno un numero di follower maggiore del numero di persone che essi seguono. Opera diversamente il metodo con parametro, che prende una lista di post ed un intero **treshold** e restituisce la lista di utenti che hanno più di **treshold** like tra tutti i post della lista.
getPosts() restituisce semplicemente la copia della lista di post di this. Serve anch’esso principalmente per **equals()**, ma è comunque una funzionalità utile da avere per analizzare una rete sociale, perciò l’ho messo public.
Come è noto **equals()** confronta due istanze di SocialNetwork (è un metodo che sovrascrive quello di Object) e restituisce true se solo se tutti i post contentuti in this e other sono gli stessi e le mappe hanno le stesse chiavi e gli stessi valori associati alle chiavi. Non solleva eccezioni se **other == null** in quanto **this != null** come garantito dall’invariante e quindi ho posto **this.equals(null) ⇒ false**.
toString() si limita a resituire una stringa che stampa la tripla nel formato che definito dalla AF.
La funzione **getMentionedUsers()**, sia con parametro che senza, restituisce la lista (senza duplicati) degli autori dei post (rispettivamente di this.posts e della lista passata come parametro).
Una nota riguardo **containing()**: ho scelto di interpretarlo come un metodo che restituisce la lista di post il cui testo contiene almeno una delle stringhe passate come parametro. Siccome il testo di un post potrebbe contenere più di una parola tra quelle della lista, nella specifica ho indicato che la collezione ritornata è un insieme, ma in realtà nell’implementazione ritorno una lista (senza duplicati), dato che il testo specifica la signature del metodo. Inoltre, il metodo aggiunge un post alla lista anche se la parola cercata compare come sottostringa di una stringa più lunga nel testo del post, in quanto il comportamento è di agevole implementazione tramite il metodo contains di String. Avrei potuto, ad esempio, sfruttare le espressioni regolari per garantire che non fossero ignorati tali casi, oppure implementare entrambe le forme del metodo. Questo è uno degli aspetti che si prestano all’estensione della classe, ma ho approfondito la creazione di sottoclassi per quanto riguarda la segnalazione di contenuti offensivi.

Estensione per segnalazione di contenuti offensivi

Una possibile estensione gerarchica della classe **SocialNetwork** che permetta di implementare la segnalazione di contenuti offensivi potrebbe consistere nel creare una sottoclasse **ModeratedSocialNetwork** che abbia come variabile d’istanza un insieme

modificabile di id, **offensivePosts**, (potrebbe essere implementato con un **Set<Integer>**) che memorizzi gli id dei post nella rete che hanno contenuti offensivi.
Per rilevarli, si dovrebbe sovrascrivere la funzione di aggiunta dei post (**addPost()** nella mia implementazione) in modo che sia eseguito anche un controllo sul testo del post, ad esempio dividendo il testo in parole (tramite il metodo **split()** con argomento opportuno) ed effettuando la ricerca di ciascuna di esse in un dizionario (**HashTable<K, V>** o simili, statico o d'istanza) di parole ritenute offensive.
Il dizionario, per ragioni di performance, dovrebbe essere inizializzato alla creazione dell'oggetto (o una volta sola se statico). Ad esempio si potrebbero leggere le parole offensive da un file e inserirle nel dizionario tramite una funzione apposita. Pertanto, quando **addPost()** è chiamata inserisce il post normalmente. Poi, se nel testo trova almeno una delle parole offensive, aggiunge l'id del post inserito all'insieme di post segnalati. Ovviamente la funzione di rimozione dei post **rmPost()** va anch'essa sovrascritta in modo che rimuova l'id del post rimosso dall'insieme di id sopracitato se era presente (ovvero se il post era stato segnalato).
Inoltre credo sia utile anche un metodo **getOffensive()** che ritorni una copia degli id dei post offensivi: ottenendo questi id un'altra classe (ad esempio Moderator, che abbia il ruolo di rimuovere post dalla rete) può implementare dei metodi che rimuovano tutti o soltano alcuni dei post offensivi contenuti. In realtà tale funzionalità potrebbe far parte anche di **ModeratedSocialNetwork** (possono anche coesistere), in modo da consentire per esempio la rimozione periodica di tutti post offensivi, che è molto utile se si vuole mettere a punto un meccanismo automatico di segnalazione e successiva rimozione dei post e si presta anche all'estensione ulteriore: potrei mantenere un contatore di post segnalati per ciascun utente della rete sociale ed eliminare tutti i post di tale autore quando il numero di quelli segnalati supera una certa soglia.
In ogni caso **ModeratedSocialNetwork** è sempre un'istanza valida di **SocialNetwork** in quanto aggiungo delle informazioni rispetto alla superclasse e nonostante sovrascriva dei metodi, essi chiamano il corrispondente metodo della superclasse ed operano modifiche sullo stato della sottoclasse (tutte le variabili d'istanza di SocialNetwork sono private, quindi non ho neanche il problema di modifiche dall'esterno).
Nel file **ModeratedSocialNetwork** ho fornito una semplice implementazione di questa sottoclasse. Ne creo un'istanza al termine del main per il testing, stampo i post contenuti (uso la stessa lista di post di MicroBlog per comodità) e stampo la lista di quelli ritenuti offensivi sulla base di una lista di parole (badwords.txt) inserite in un dizionario statico.

Testing

Ho scritto una classe Test contenente il metodo main con il codice per il testing: il programma legge da standard input un testcase ed esegue varie stampe ed assert, pertanto è sufficiente digitare su terminale (nella directory che contiene tutti i file del progetto):

```
#       javac *.java
```

per compilare tutte le classi, e poi

```
#       java -ea Test < test?.txt
```

per eseguire il testcase scelto (-ea è serve per abilitare le asserzioni). Sono presenti sei testcase che prendono in esame varie situazioni che possono verificarsi in istanze di Post e di SocialNetwork.
Leggo una lista di post dal testcase e poi leggo una lista di like (sempre presi dal testcase) che aggiungo a post scelti a caso tra tra quelli della lista. Questo segmento di codice potrebbe sollevare l'eccezione SelfLikeException, che è gestita con un blocco try-catch. Una volta aggiunti tutti i like inizializzo MicroBlog con la lista di post; poi creo una rete sociale vuota ed aggiungo tutti i post della lista, verificando che prima le

due reti siano diverse e poi uguali. In seguito creo intenzionalmente un post con id uguale al primo della lista di post della rete sociale (se ho almeno un post) e tento di aggiungerlo, sollevando l'eccezione DuplicatePostException. Successivamente svuoto la seconda rete (devono quindi risultare diverse) e provo anche a togliere da MicroBlog un post che sicuramente non è presente (lo costruisco con id=-1 e, a causa del modo in cui genero gli id nel costruttore usuale di Post ,ciò non può avvenire) quindi solleva NoSuchPostException. Tramite assert e stampe testo gli altri metodi della classe, tra cui containing(), a cui passo la lista di parole letta dal testcase. Infine creo un'istanza (usando la stessa lista di post di MicroBlog) della rete sociale moderata ModeratedSocialNetwork, che implementa la segnalazione, e stampo gli id post segnalati.
L'output che dovrebbe risultare dall'esecuzione è in parte fisso ed in parte casuale (l'assegnazione degli id e dei like ai post, ma di conseguenza anche lo stato di SocialNetwork); invece il testo dei post, i nomi degli utenti che mettono like e le parole da ricercare (anche le badword), sono fissi.

Una piccola nota riguardo la stampa dei Set: nei metodi in cui era necessario fornire la rappresentazione dei Set usati come stringhe (toString() che ho sovrascritto sia in Post che in SocialNetwork) ho preferito usare il metodo predefinito toString(), che restituisce la stringa [el_1, ..., el_n], però differisce dalla usuale notazione matematica con le parentesi graffe, tuttavia è indicato nei commenti quali sono insiemi e quali liste.