Dipartimento di Informatica, Corso di Laurea in Informatica
Relazione primo progetto intermedio del corso di Programmazione II, a.a. 2020-21
Docente di riferimento: Gianluigi Ferrari
Salvatore Correnti, matricola 584136

Il progetto richiede l'implementazione di una componente software per la gestione di un social network nel quale gli utenti possono pubblicare post testuali, con un massimo di 140 caratteri. È possibile inoltre "seguire" i post di altri utenti: nella rete questo viene implementato attraverso la possibilità di "aggiungere like" ai post degli altri utenti. Inoltre, come specificato nelle consegne, gli utenti non possono seguire se stessi e di conseguenza non è possibile aggiungere like a un proprio post.
Si può immaginare la rete come una community in cui tutti gli utenti hanno la possibilità di aggiungere o rimuovere post, mettere like a post di altri utenti e di conseguenza seguirli.
Il progetto si compone di 3 interfacce e 11 classi (inclusa la classe di test TestAll.java e la *utility class* Pair.java). Per meglio organizzare l'insieme dei sorgenti si è deciso di ripartirli in 4 *packages*:

- **microBlog.network** contiene SocialNetwork, MicroBlog, MicroBlogExt, Pair e PermissionDeniedException;
- **microBlog.user** contiene User, UserImpl e UserException;
- **microBlog.post** contiene Publishable, Post, TextPost, Like, Tag e PostException;
- **microBlog.test** contiene TestAll.

Nel seguito ogni interfaccia e classe è descritta nel dettaglio, insieme alle principali scelte di progetto.

Interface User

L'interfaccia User rappresenta un generico utente della rete sociale. Un utente è un oggetto che può pubblicare / rimuovere post e seguire (i post di) altri utenti. Ogni utente è univocamente definito dal proprio *username*, il quale è una sequenza di caratteri alfanumerici priva di spazi e che comincia con una lettera. Ad ogni utente è altresì associata la sua data di registrazione alla rete.

L'aggiunta del tipo di dato User al progetto ha comportato la necessità di creare un meccanismo di sicurezza che impedisca a un'entità "esterna" all'utente stesso di modificarne i dati presenti sulla rete. Poiché l'applicazione non prevede un'interfaccia grafica e User e SocialNetwork convivono in *packages* diversi, si è deciso di implementare ciò attraverso un meccanismo di *richiesta* di modifica dei dati alla rete da parte dello stesso oggetto di tipo User: ogni volta che egli intende aggiungere / rimuovere un post o aggiungere un like, fa esplicita richiesta di ciò alla rete a cui è registrato attraverso l'opportuno metodo; l'interfaccia SocialNetwork definisce i corrispettivi metodi che permettono di controllare se effettivamente egli ha fatto richiesta invocando il metodo <User>.hasSentRequest(), e se questi restituisce *true* allora effettuano tali modifiche, altrimenti lanciano l'eccezione *PermissionDeniedException*.
Per garantire inoltre una maggior sicurezza e semplificare la fase di test si è inoltre deciso di evitare che l'utente adoperi direttamente attraverso oggetti di tipo *Post*: tutti i metodi suddetti accettano in input *soltanto* interi e stringhe, che identificano rispettivamente *id* di Post e Like e *username* degli utenti. L'id e l'autore di ogni post restano infatti costanti, mentre il riferimento a un post può facilmente diventare invalido durante la fase di test a causa dell'implementazione.
Note sui metodi:

- **writePost**: accetta in input il testo del post;
- **removePost:** accetta in input l'id del post;

- **addLike**: accetta in input l'id del post e lo username del suo autore per i dovuti controlli;
- **hasSentRequest**: restituisce true se e solo se l'utente ha inoltrato una richiesta alla rete.

Class UserImpl implements User

La classe UserImpl implementa l'interfaccia User. Ogni oggetto di questa classe contiene internamente un riferimento alla sua rete sociale, ed il campo *sentRequest* memorizza il valore di verità dell'invio di una richiesta alla medesima.

Note sui metodi:

- **UserImpl** (costruttore): prende in input un riferimento a una rete sociale e vi registra automaticamente il nuovo utente;
- **writePost, removePost, addLike**: per implementare i controlli di sopra, si è scelto di porre sentRequest = true subito prima dell'invocazione del metodo corrispondente della rete e = false subito dopo, cosicché non vale mai *true* all'infuori del corpo di un metodo.

Interface SocialNetwork

Rappresenta la rete sociale definendo:

- una serie di metodi *modificatori* (**storePost, removePost, registerUser, addLike**) che gestiscono le relazioni fra utenti / post / like. Come già detto, ognuno di questi lancia un'eccezione se non chiamato dal corrispondente utente (autore del post / like, utente in fase di registrazione). Questi metodi prendono in input oggetti di tipo Post / User, opportunamente ottenuti o dallo stesso utente chiamante o attraverso il metodo **getPost**;
- una serie di metodi *osservatori* (**containsPost, isRegistered, isLikedByUser, isFollowing, getTotalFollowersCount**) che restituiscono valori interi o booleani e sono delle query sui dati interni della rete;
- una serie di metodi *produttori* (tutti i restanti) che restituiscono copie sicure di dati presenti nella rete sociale (e.g., una lista dei post scritti da un determinato utente etc). Tali metodi preservano l'information hiding in quanto non forniscono alcun riferimento ad eventuali strutture implementative delle sottoclassi, mentre i Post / Like / Tag restituiti *possono* riferirsi agli stessi presenti nell'implementazione in quanto tali tipi di dati sono immutabili.

Note sui metodi:

- **getPost**: restituisce il Post con id e username dell'autore forniti se presente, ed è pensato per permettere all'utente di non operare direttamente con i Post / Like, e per fornire un semplice accesso a ogni post in fase di test.

Class MicroBlog implements SocialNetwork

La classe MicroBlog estende l'interfaccia SocialNetwork. Come struttura di implementazione si è adoperata:

- Map<String,Set<String>> network, in cui per ogni utente a, Map[a] è l'insieme degli utenti seguiti da a;
- Map<String,Set<Post>> posts, in cui per ogni utente a, Map[a] è l'insieme dei post pubblicati da a;
- List<Pair<String>> followersCount, una lista ordinata in senso decrescente di coppie (intero, stringa) per mantenere il conto dei followers di ogni utente.

Note sui metodi:

- **writtenBy, containing**: le List<Post> restituite non hanno alcun collegamento con il Set<Post> interno.

Pair<T> implements Comparable<T>

La classe Pair è una semplice classe di supporto che rappresenta coppie di oggetti di tipo (int, T), usata per tenere lista ordinata decrescente del numero di followers di ogni utente per supportare il metodo influencers() di SocialNetwork. Non è visibile al di fuori di microBlog.network .

Interface Post

Rappresenta un Post pubblicabile sulla rete sociale. Ogni Post ha associato un Set<Like> che rappresenta i suoi likes e un Set<Tag> che rappresenta gli utenti taggati nel testo.
L'interfaccia contiene inoltre due metodi statici utilizzati per controllare la correttezza del testo di un post. Come scelta di progetto si è deciso di ignorare i marcatori "@" dei tag nel calcolo della lunghezza del testo, in quanto non sarebbe necessario indicarli esplicitamente ad esempio in una stampa a schermo una volta memorizzato il nome dell'utente taggato.
Come riportato nelle consegne, il tipo di dato Post è immutabile.
Note sui metodi:

- **copy**: metodo utilizzato per "aggiornare" un post all'aggiunta di un like: crea una "copia" del post vecchio in cui sostituisce l'insieme di likes con quello passato in input;
- **(static String) removeTagSignatures**: restituisce il testo passato in input privo dei marcatori dei tag;
- **(static boolean) checkTextLength**: controlla se il testo restituito da removeTagSignatures ha non più di 140 caratteri.

Class TextPost implements Post

Implementazione di Post, che rappresenta gli oggetti che ogni istanza della classe UserImpl può effettivamente inserire in MicroBlog. Come riportato fra le richieste, ogni post della rete utilizza un identificativo univoco: per essere in grado di assegnare univocamente tale id, si è deciso di utilizzare un campo statico finale currentMaxUsedId che tiene traccia de nuovi post creati, in modo tale che l'id dell'n-esimo nuovo post creato dall'avvio dell'applicazione sia esattamente n (di conseguenza ogni id è >= 1).
Per tenere traccia dei tag e dei like si è deciso di implementare un campo likes e uno tags. Si è pertanto deciso di aggiungere un secondo costruttore privato utilizzato quando viene aggiunto un nuovo like, dato che Post è immutabile. Come già detto, con questo costruttore viene creato un nuovo TextPost copia del precedente, tranne per this.likes che viene inizializzato a this.getLikes().add(<nuovo_like>).
Come specificato in SocialNetwork e in User, le istanze della classe possono essere inserite o rimosse in un oggetto di tipo SocialNetwork solamente se richiesto dallo User autore del post.

Class Tag implements Cloneable

Un Tag nel testo ha la forma @<username>, dove username è definito come specificato in User, e deve chiaramente essere lo username di un utente effettivo della rete. L'insieme dei Tag nel testo di un TextPost è aggiunto come private Set<Tag> tags.
Dato che, a differenza dei Like, i Tag non possono essere aggiunti e modificati in quanto il testo di un Post è immutabile, non serve che siano passati come parametro al "costruttore di copia" in TextPost.

Class Like

Rappresenta un like ad un Post. Essendo un'implementazione di Publishable possiede i campi autore, data e codice identificativo, oltre ad un riferimento al Post a cui appartengono e ad un riferimento all'utente del post che ha ricevuto il like (*receiver*).

Class MicroBlogExt extends MicroBlog

Estensione della classe MicroBlog in cui alcuni contenuti testuali offensivi possono essere segnalati. Il costruttore permette di passare una lista di parole che sono considerate offensive all'interno della rete. Si invita a notare che i Post che contengono queste parole non vengono rimossi bensì si segnala semplicemente il contenuto offensivo. Attraverso il metodo isOffensive si può inoltre chiedere se un determinato post è stato segnalato dalla rete come offensivo.

Eccezioni ridefinite

Nel progetto sono presenti 3 eccezioni ridefinite, tutte unchecked: *PermissionDeniedException, PostException, UserException.*

- **PermissionDeniedException:** classe di eccezione unchecked, utilizzata in ogni caso di tentativo di accesso illegale alla rete (ad esempio la modifica non autorizzata di post etc);
- **PostException:** classe di eccezione unchecked, utilizzata nel caso in cui sia dato in input un oggetto di tipo Post non "corretto" (e.g., lunghezza del testo superata o tentativo di inserimento in una rete sociale a cui l'autore non appartiene);
- **UserException:** classe di eccezione unchecked, utilizzata nel caso in cui sia dato in input un oggetto di tipo User non "corretto" (e.g., username non valido o con spazi al suo interno oppure non corrispondente all'autore di un post etc).

Batteria di Test

Le due batterie di test sono state implementate in modo da garantire la stampa e la cattura di ogni eccezione che ogni metodo del progetto può lanciare. Per fare ciò si è usata una serie di try-catch, ognuno dei quali corredato con un commento esplicativo dell'errore effettuato e con un esempio di esecuzione corretta.
La prima batteria di test contiene tutti i test necessari ad esclusione di quelli di MicroBlogExt, i quali sono invece presenti nella seconda batteria per mostrare il corretto funzionamento del meccanismo di segnalazione di contenuti offensivi.

Documentazione

La directory /MicroBlog/doc contiene la documentazione del progetto realizzata attraverso il tool Javadoc; è possibile navigare fra le informazioni delle varie classi attraverso il file *allclasses-index.html*.
Sono stati aggiunti alcuni *custom tags* per visualizzare correttamente nella specifica le clausole *requires, modifies, effects, overview, rep-invariant, typical element, abstract function*.
In caso di necessità, lo script javadoc.xml consente di rigenerare la documentazione applicando di default tali impostazioni.
NOTA: Dato che Javadoc accetta html in input, i simboli "<", ">", "<=", ">=", "&" non possono essere inseriti nelle specifiche dei metodi direttamente nei sorgenti, ma devono essere sostituiti dai corrispondenti caratteri html: "<", ">", "≤", "≥", "&". È chiaramente possibile però leggere tutto correttamente dalla documentazione.

Avvio

Per avviare l'applicazione basta lanciare il file TestAll.class che contiene i due metodi statici di test.