Programmazione II - Progetto Java

Lorenzo Leonardini - matricola 598608

Compilazione ed esecuzione

Il progetto è stato realizzato principalmente con Eclipse, nell'archivio consegnato vi sono i file contenuti nella cartella `src`. Per eseguire la batteria di test si può creare un progetto Eclipse importando il codice oppure, più semplicemente, si può sfruttare il Makefile allegato contenente le regole `all`, `test`, `clean` rispettivamente per generare il jar, eseguire i test e ripulire l'ambiente di lavoro.

Se si decide di importare il progetto in un IDE, notare che la classe `Test`, facendo ampio utilizzo della funzione `assert`, richiede l'aggiunta della flag `-ea` agli argomenti della VM.

Eccezioni "checked" vs. "unchecked"

Il progetto fa ampio utilizzo di eccezioni personalizzate. Il criterio con cui si è scelto quando estendere `Exception` o `RuntimeException` è semplice: se la "responsabilità" dell'errore è del chiamante (in generale quando si tratta di passare parametri validi nel formato corretto), allora le eccezioni sono di tipo unchecked, in quanto si ha la possibilità di controllare la validità dei dati prima dell'invocazione; se invece l'eccezione è generata da uno stato interno che il chiamante potrebbe non conoscere, allora si utilizza il tipo checked, in modo da garantire che ogni casistica sia adeguatamente gestita.

La classe `Post`

La classe `Post` è una semplice classe immutabile contenente i dati di un post pubblicato nella rete sociale. L'invariante di rappresentazione coincide con la validità dei dati (valori non nulli e che rispettano i requisiti di base). Sebbene l'oggetto non debba far altro che racchiudere alcune variabili, si è scelto di aggiungere un po' di logica per il parsing del contenuto del post, in modo da alleggerire e ottimizzare il lavoro della classe `SocialNetwork`. Prima di prendere in analisi questi metodi, però, è necessario soffermarsi su alcune scelte di implementazione.

La prima, più semplice, riguarda il timestamp. La consegna del progetto non specifica in modo particolare che tipo di variabile debba essere `timestamp`, se non il fatto che debba contenere le informazioni riguardanti la data e l'ora di creazione del post. Personalmente preferisco, quando possibile, utilizzare timestamp UNIX (o in generale timestamp numerici): permettendo di visualizzare date e orari con fuso locale e in modo "relativo" (es. "1 ora fa") offrono una maggiore flessibilità. Il timestamp viene quindi calcolato dalla funzione `System.currentTimeMillis()`. Se si desidera avere rappresentazioni più ad alto livello, si sono aggiunti due metodi getter, `getPublishedDate()` e `getFormattedDate()`, che ritornano rispettivamente un'istanza dell'oggetto `Date` e una stringa in formato "dd/mm/yyyy hh:mm".

La generazione di un id univoco è implementata con il supporto di una variabile statica, `base_id`, che viene incrementata alla creazione di ogni post. L'id corrente può quindi assumere il valore del contatore, con la garanzia che post passati e futuri avranno id diversi. Chiaramente bisogna tenere in considerazione le limitazioni del tipo di dato `int`: in base alle proprie necessità potrebbe essere cambiato in `long` o `BigInt`, ma al fine di questo progetto non sembrava necessario.

Un po' più di libertà sono state invece prese per l'username dell'autore. Nel tentativo di ricercare un'implementazione tutto sommato realistica, si è deciso di considerare l'username

come fosse quello di twitter o instagram: un'unica parola alfanumerica che identifica in modo univoco un'utente. In particolare (principalmente per semplificare il parsing), tutti gli username devono rispettare la regex `^[0-9a-zA-Z]{4,}$`. Nella realtà, però, gli username sono case insensitive: taggare `@user` e `@USER` è la stessa cosa, e quest'aspetto voleva essere inserito anche nel progetto. Tuttavia, per evitare confusione, si è scelto di considerare "ufficiale" solo l'username lowercase. Con "ufficiale", in questo caso, s'intende il nome utente memorizzato come autore di un post e contenuto nella mappa dei follower. Gli username nel testo dei post, invece, non vengono modificati.

L'ultimo dato ancora da predere in esame è appunto il contenuto dei post. Come prima cosa viene controllata la sua validità in termini di lunghezza: il testo deve essere almeno lungo un carattere e al massimo lungo 140. Prima del controllo, sempre a scopo di realismo, viene chiamato il metodo `strip`, in modo da eliminare spazi in cima e in fondo al messaggio. Per parlare del parsing, invece, bisogna prima analizzare come si è scelto di implementare la meccanica dei follow: un utente può seguire qualcun altro o taggandolo oppure menzionando un suo post. Ispirato dagli issue su GitHub, le due meccaniche vengono implementate con la chiocciola per gli utenti (`@user`) e con l'hash per i post (`#id`) (es. "Menziono il post #2 di @admin"). Il parsing diventa quindi quasi immediato grazie all'aiuto delle regex. Il risultato viene salvato per uso futuro in due Set: `people_tagged` e `post_mentions`, che vengono resi read-only mediante l'invocazione di `Collections.unmodifiableSet`. In questo modo possono essere ritornati direttamente da due metodi getter, senza la possibilità che vengano effettuate delle modifiche e senza la necessità di copiarli in nuovi Set ogni volta per garantire l'immutabilità della classe. Il parsing è anche il primo momento in cui si verifica il controllo della validità dei post menzionati: non si può chiaramente fare riferimento a un Post futuro, nel qual caso viene generata un'eccezione.

La classe `SocialNetwork`

Data la lista di metodi richiesti, risulta palese che la struttura dati principale della classe, prima ancora della mappa dei follow, sia un qualche tipo di lista contente i vari post da cui estrarre informazioni. Sebbene sia vero che molti metodi prendono una lista di post come parametro, infatti, è anche vero che altri operano sull'intera rete sociale e non richiedono un sottoinsieme di post, per cui è necessario avere un modo per memorizzare tutti i contenuti del social network.

Il primo approccio sarebbe quello di utilizzare una semplice `List`, su cui chiamare `list.add()`. Tuttavia, sarebbe utile poter utilizzare a proprio vantaggio l'id numerico: inserendo un post con id i in posizione i della `List`, infatti, si ha la possibilità di accedervi direttamente senza doverlo cercare ogni volta. Il problema di questa soluzione, però, è che quando un post non viene aggiunto alla rete sociale, al suo posto bisognerebbe inserire un qualche tipo di padding, in modo che la proprietà venga mantenuta. In primo luogo questo comporterebbe un inutile spreco di memoria. Inoltre utilizzare il valore `null` come padding richiederebbe anche diversi controlli quando si estrae un post dalla lista.

La soluzione adottata prevede invece l'utilizzo di una `Map<Integer, Post>`. Questo non solo si basa sulla proprietà per cui i post hanno id univoco, ma la rafforza anche: utilizzando una mappa è impossibile che due post con il medesimo id siano contenuti nella stessa istanza di `SocialNetwork`.

Il primo metodo extra aggiunto alla classe è necessariamente il metodo `void addPost(Post p)`, indispensabile per aggiungere un post alla mappa. In questo metodo vengono anche effettuati alcuni controlli per la validità del post che si desidera aggiungere. Uno di questi verifica che i post menzionati siano contenuti nella rete sociale. Questo controllo, unitamente a quello svolto alla creazione del post, garantisce il corretto funzionamento degli altri metodi. In questa occasione viene anche aggiornata la mappa dei follow. Un'operazione che, una volta effettuato il parsing del post, è particolarmente immediata.

Il metodo `Map<String, Set<String>> guessFollowers(List<Post> ps)` non è quindi particolarmente complesso. Viene inizializzata la mappa, poi per ogni post vengono aggiunte le

informazioni sui follower che questo offre, facendo attenzione a creare i `Set` quando necessario e appurandosi di far sì che un utente non possa seguire sé stesso.

Il metodo `List<String> influencers(Map<String, Set<String>> followers)`, per poter ritornare la lista di utenti ordinata in base al numero di follower, però, necessiterebbe di una mappa strutturata nel modo inverso: non servono le informazioni sui follow, ma sui follower. Il primo passo è quindi quello di utilizzare la mappa passata come parametro per computare una seconda mappa le cui chiavi siano gli username e i valori siano il numero di follower (di fatto non servono altre informazioni). A quel punto basta ordinare le chiavi per i valori. Grazie alle librerie interne di Java, basta utilizzare `map.entrySet()`, `list.sort()` e `Entry.comparingByValue()`. Notare che in questo modo la lista ritornata non contiene gli utenti con zero follower, ma d'altronde, trattandosi di una funzione con lo scopo di estrarre gli "influencer" della rete sociale, ha perfettamente senso.

Il metodo `Set<String> getMentionedUsers(List<Post> ps)` è di realizzazione immediata: poiché ogni istanza di `Post` contiene già gli utenti menzionati, basta scorrere ogni post nella lista e aggiungere tutti gli username ad un `Set`.

Anche il metodo `List<Post> writtenBy(List<Post> ps, String username)` è semplice: scorrendo tutta la lista di post si estraggono quelli che hanno l'autore uguale all'username cercato (con l'accortezza di utilizzare il metodo `equalsIgnoreCase`).

Per il metodo `List<Post> containing(List<String> words)` si è scelto di non cercare parole precise, ma di fare un semplice `String.contains`. Si tratta di una pura preferenza personale, principalmente per permettere di cercare parole senza coniugazioni o declinazioni precise. Sempre per un simile motivo, si è scelto di implementare una ricerca case insensitive.

Tutti i metodi senza lista di post tra i parametri chiamano il corrispettivo metodo avvalendosi di `List<Post> getPosts()`, che si assicura di creare una nuova lista, in modo da non esporre l'implementazione interna della classe.

I metodi in cui viene passata una lista di post tra i parametri, invece, presuppongono che questa lista sia un sottoinsieme dei post contenuti nella rete sociale. A dire il vero non si tratta di una vera e propria necessità, se non per il metodo `guessFollowers`, che specifica che uno dei casi in cui viene lanciata l'eccezione `NullPointerException` è quando, per l'appunto, uno dei post della lista non è contenuto nella map globale. L'eccezione viene lanciata dalla chiamata a `posts.get(post_id).getAuthor()` all'interno del metodo `updateNetwork`. Questo significa che utilizzare una lista che non è sottoinsieme di `posts` non porta direttamente al sollevamento di un'eccezione, anzi la computazione potrebbe andare a buon fine. L'eccezione è invece parte del comportamento indefinito che si verifica in seguito alla violazione della clausola `@requires ps is a subset of the posts added to SocialNetwork`. Perché non svolgere dei controlli e causare un errore ogni volta che la clausola non è soddisfatta? Perché lasciare un comportamento indefinito? Poiché la violazione della clausola non porta a danni nella struttura interna della classe e poiché il controllo richiederebbe un certo overhead (minimo e che non aumenta la complessità della funzione, ma inutile), si ritiene che la scelta migliore sia quella implementata.

L'invariante di rappresentazione di `SocialNetwork` è necessariamente più complessa di quella di `Post` e tra le varie cose comprende:

- $\forall\ a\ \ network[a] = \{b | \forall\ p \in posts \text{ t.c. } p.author = a,\ b \in p.mentioned\_users \text{ or } b \in p.mentioned\_posts.author\} \setminus \{a\}$
- $\forall\ i\ \ posts[i].getId() = i$
- $\forall\ i\ \ \forall\ j \in posts[i].getPostMentions(),\ post[j]$ è valido*

*con valido s'intende che la chiave in questione esiste nella mappa e il valore associato è diverso da `null`.

In generale, per quanto riguarda il controllo dei valori in input e output, liste e mappe vuote sono sempre considerate valide. In questo modo, qualsiasi valore di output può essere utilizzato

come valore di input di un altro metodo senza necessitare alcun tipo di controllo. I metodi possono quindi essere chiamati "a cascata" (es. `influencers(guessFollowers(containing(s)))`), semplicemente verificando la validità del parametro iniziale (che nella maggior parte dei casi basta non sia `null`).

Eliminazione dei Post

La possibilità di eliminare post può essere implementata in diversi modi. Alcuni di questi sono presi in esame con rispettivi pro e contro.

L'approccio più semplice consiste semplicemente nell'eliminare il post dalla rete sociale. Tuttavia questo andrebbe contro l'invariante di rappresentazione, in particolare contro il punto che prevede che ogni post menzionato sia presente nella rete: se si dovesse eliminare un post citato da qualcun altro, la classe non si comporterebbe come dovrebbe. Si potrebbe cambiare l'invariante di rappresentazione nella sottoclasse `DeletableSocialNetwork`, ma ciò richiederebbe di reimplementare tutti i metodi tenendo conto di questa eventualità, andando a vanificare il senso di una sottoclasse (a quel punto sarebbe forse meglio utilizzare un'interfaccia). Ovviamente non si tiene conto della possibilità di riscrivere `SocialNetwork`, non avrebbe senso.

Inoltre questa soluzione va contro a come personalmente preferisco implementare l'eliminazione: quando un post viene eliminato, l'unica informazione che si vuole perdere è il contenuto offensivo o che in generale viola le condizioni del social network. Autore e data vogliono invece essere mantenuti, così come l'id. Questo in particolare pensando ai thread: quando si crea una discussione attorno a un post che poi viene cancellato, non si vuole che i link delle menzioni "si rompano", si vuole invece mostrare un messaggio "post eliminato". Mantenere autore e data permette di dare un certo contesto al thread, permettendo a chi legge nel futuro di capire quando e da chi è partito tutto.

Il modo più semplice per implementare un sistema del genere richiede un supporto da parte della classe `Post` stessa. Questa non sarebbe più immutabile e comprenderebbe i metodi necessari per la segnalazione e l'eliminazione. Tuttavia il testo del progetto richiede che quest'espansione del social network venga implementata estendendo la classe `SocialNetwork` e non la classe `Post`. Si potrebbe pensare di risolvere creando sia `DeletableSocialNetwork`, sia `DeletablePost`, ma questo presenta due problemi. Il primo riguarda creare una sottoclasse mutabile di una classe immutabile: sebbene il linguaggio lo permetta, non si tratta certamente di una buona scelta. Il secondo: far si che il parametro di `DeletableSocialNetwork.addPost` sia un'istanza di `DeletablePost` (lanciando un'eccezione altrimenti) vanifica ancora una volta il senso di avere una sottoclasse.

La soluzione finale implementata consiste nel sostituire un post eliminato con un'istanza della classe `DeletedPost`. Questa è una sottoclasse di `Post` e viene creata a partire da un'istanza di quest'ultimo, creandone una copia, ma censurandone il testo (rimpiazzandolo con "POST DELETED"). Quest'approccio presenta alcune criticità. La più banale è la necessità da parte della classe `Post` di creare un'istanza con id e timestamp personalizzati. Per ovviare al problema si è quindi aggiunto un costruttore `protected`. La seconda criticità riguarda appunto la possibilità di creare un post con id personalizzato: questo non solo permette la creazione di post con id non validi (per esempio negativi), ma rompe la proprietà per cui un post ha id univoco. Il primo problema non è particolarmente rilevante: trattandosi di un costruttore `protected` si ha il controllo dei parametri con cui viene chiamato. Il secondo invece potrebbe sembrare più critico, ma bisogna ricordare che salvando i post in una `Map<Integer, Post>` si ha la garanzia che all'interno di un'istanza di `SocialNetwork` non ci possano essere due post con lo stesso id, che quindi per estensione è univoco. Per quanto ci possano essere due post con il medesimo id in memoria, questi post non possono convivere nella stessa rete sociale.

In realtà si può individuare un ultimo problema: un utente può infatti "salvarsi" un post prima che questo venga eliminato. In seguito alla cancellazione il post scomparirebbe dal social network ma rimarrebbe accessibile a chi lo ha estratto precedentemente (con metodi quali

`getPosts()`). Tuttavia si tratta di un problema impossibile da risolvere, a meno di non avere un post mutabile su cui poter operare direttamente. Anche la soluzione che prevede la rimozione del post dalla struttura dati (rimpiazzandolo con `null`) è soggetta alla stessa problematica, che quindi non può essere affrontata in alcun modo.

Per quanto riguarda il sistema di segnalazione e rimozione, un utente ha la facoltà di segnalare un post ritenuto offensivo. Successivamente un "admin" ha la possibilità di estrarre tutti i post segnalati (più segnalazioni ci sono per un determinato post e più importanza gli viene conferita) e può scegliere se eliminarlo oppure se approvarlo e far si che non possa più essere segnalato. Per far ciò vengono aggiunte due strutture dati: una lista di post approvati e una mappa che lega ogni post segnalato al numero di segnalazioni ricevute.

Notare come venga introdotto il concetto di admin, che non fa parte né di `SocialNetwork` né di `DeletableSocialNetwork`. Poiché però il social network non ha neanche il concetto di registrazione e login, l'idea è che per la gestione degli utenti (e dei loro ruoli) venga utilizzata una libreria esterna, che si occupa tra le altre cose di definire l'implementazione di un utente admin. Sarà poi responsabilità del chiamante verificare che determinati metodi critici vengano resi accessibili solo agli amministratori e moderatori.

Alcune informazioni sull'implementazione: è importante che il metodo `delete(Post p)` ricalcoli la mappa dei follower, in quanto `DeletedPost` perde ogni tipo di menzione sia a utenti sia a post. La soluzione più semplice consiste nell'aggiungere `network = guessFollowers(getPosts())`.

Il metodo `reviewPost` ritorna i post segnalati ordinandoli in base al numero di segnalazioni ricevute, utilizzando di fatto le stesse meccaniche con cui si sono ordinati gli utenti per numero di follower nel metodo `influencers`.

L'ultimo aspetto su cui è il caso di concentrarsi brevemente è l'invariante di rappresentazione. Alle condizioni standard della classe `SocialNetwork` si sono aggiunte:

- $\forall\ i \in approved\_posts\ \ posts[i]$ è valido
- $\forall\ [i,j] \in reports\ \ post[i]$ è valido e $j > 0$
- $\forall\ i \in approved\_posts\ \ i \notin reports$ e $\forall\ [i,j] \in reports\ \ i \notin approved\_posts \Rightarrow$ approved_posts e reports sono in mutua esclusione