Primo progetto intermedio

Samuele Bonini

Sommario

Relazione e documentazione della soluzione presentata al primo progetto intermedio del corso di Programmazione 2.

Indice

1 La classe Post

Viene richiesta l'implementazione di una classe **Post**, utilizzata per rappresentare un generico post pubblicato su una rete sociale.

1.1 Variabili d'istanza

1. **Identificatore**. La variabile `id` è un intero non negativo che identifica univocamente un'istanza della classe Post *nell'ambito di una rete sociale*. L'unicità dell'identificatore è garantita dalla classe **SocialNetwork**,

in quanto la classe Post non ha sufficienti informazioni per incapsulare il meccanismo che fornirebbe tale garanzia. Il valore dell'identificatore non dev'essere modificato una volta che l'istanza di Post viene creata, pertanto la variabile è contrassegnata come `final`.

2. **Nome dell'autore**. Stringa che identifica il nome utente dell'autore del post. Non può essere vuota o composta da soli spazi. Quest'ultima condizione viene espressa dicendo che la stringa non deve appartenere al linguaggio definito dall'espressione regolare `/^\s+$/`. Anche la variabile `author` è contrassegnata come `final` in quanto non modificabile dopo l'istanziazione.

3. **Contenuto del post**. Stringa di lunghezza compresa tra 1 e 140, modificabile dal metodo `editPost()`.

4. **Data e ora di pubblicazione**. Variabile di classe `Timestamp` istanziata al momento della creazione di un oggetto mediante una chiamata a `System.currentTimeMillis()`. Non è contrassegnata come `final` in quanto, sebbene non modificabile direttamente dall'utente, deve essere modifica dal metodo privato `setTimestamp()` quando viene effettuata una chiamata al metodo `clone()` di Post (che, altrimenti, non creerebbe correttamente una copia del post in quanto il costruttore di default istanzia il timestamp al tempo corrente).

5. **Lista degli utenti che hanno messo like al post**.
`LinkedList` di stringhe contenente i nomi di tutti e soli gli utenti che hanno messo like al post.

1.2 Metodi di rilievo

In aggiunta ai metodi *getters* e *setters*, la classe dispone di metodi per: clonare un oggetto, ottenere una rappresentazione dell'oggetto come stringa, aggiungere e rimuovere like al post, e infine comparare due istanze di Post.

Per quanto riguarda l'ordinamento, in accordo con la semantica di `id`, vale la seguente relazione:

$$\forall p, q \text{ post } .\ p = q \iff p.getId() = q.getId()$$

Questa legge di ordinamento opera sotto l'assunzione che due post vengano confrontati solo se appartenenti alla stessa rete sociale (in quanto è quest'ultima a garantire l'unicità degli id).

Per esplicitare ulteriormente questa condizione, una possibile modifica all'implementazione del progetto potrebbe consistere nello spostare la classe

Post all'interno dell'implementazione della classe **MicroBlog**, trasformandola quindi in una *nested class*.

1.3 Funzione di astrazione ed elemento tipico

La funzione di astrazione della classe Post è essenzialmente la funzione identità. Un elemento della classe può essere rappresentato come una 5-tupla contenente le variabili di stato dell'oggetto.

L'attributo `likes`, sebbene implementato nel concreto come una lista, mappa in realtà su un insieme nell'astratto, in quanto le pre-condizioni dei metodi che aggiungono o rimuovono like dalla lista garantiscono che i like siano unici per post. Pertanto, ogni istanza concreta dell'attributo `likes` verrà mappata a uno stato astratto nel seguente modo:

$$[like_1, ..., like_n] \mapsto \{likes.get(i) \mid 0 \leq i < n\}$$

1.4 Dimostrazione di correttezza della classe

Segue la dimostrazione che la classe rispetta l'invariante contenuta nel file `Post.java`.

Dimostrazione.

- **Caso base**: Il seguente frammento di codice all'interno del costruttore garantisce le condizioni descritte dall'invariante per gli attributi `text` e `author`:

```
1 if(id < 0 || author.trim().isEmpty() || text.trim().
     isEmpty()) {
2     throw new IllegalArgumentException();
3 }
4 if(text.length() > 140) {
5     throw new LimitExceededException();
6 }
7
```

- **Passo induttivo**: Il metodo `editPost()` garantisce le condizioni sulla variabile d'istanza `text`, mentre il metodo `addLike()` le garantisce sui contenuti della lista `likes`. `setTimestamp()` verifica che il parametro passato non sia `null`. Gli altri membri sono `final` e quindi non modificabili.

□

2 La classe MicroBlog

La classe implementa l'interfaccia **SocialNetwork** definita nella specifica del progetto. Viene inoltre definita la relazione di "follower": un utente ne segue un altro se e solo se è registrato (ha scritto almeno un post) ha messo like ad almeno un post scritto da quell'utente. Un utente non registrato può comunque mettere like ai post, ma non verrà conteggiato tra i follower degli utenti a cui ha messo like.

2.1 Strutture dati

Sono utilizzate 3 strutture dati per lo *storage* interno dello stato:

- `followRelations`: mappa stringhe su insiemi di stringhe. Ogni chiave identifica univocamente un utente registrato nella rete. Un utente è registrato nella rete se e solo se ha almeno un post all'interno della rete (si veda la struttura dati successiva). Per ogni utente u, l'insieme su cui u è mappato contiene tutti e soli gli utenti seguiti da u.

- `postRelations`: mappa stringhe su insiemi di Post. In ogni istante, l'insieme delle chiavi di questa struttura è identico a quello delle chiavi in `followRelations`, ovvero l'insieme degli utenti registrati alla rete. Se u è un utente, allora l'insieme $followRelations.get(u)$ è quello di tutti e soli i Post scritti da u.

- `postLookup`: mappa interi su Post. Questa struttura permette la ricerca di Post per id in tempo $O(1)$ al caso medio, permettendo un accesso ai dati in tempo ottimizzato, in quanto l'assenza di questa struttura richiederebbe la scansione di tutta la struttura `postRelations` per trovare un Post, il che richiederebbe tempo $O(n)$.

2.2 Metodi di rilievo

La classe contiene diversi metodi oltre a quelli presenti nella specifica dell'interfaccia. Si riportano di seguito informazioni su quelli più di rilievo.

Il metodo `guessFollowers()` e le versioni di `writtenBy()`, `getMentionedUsers()` e `influencers()` che hanno firma non vuota sono indipendenti dallo stato. Pertanto, sono definiti come metodi **statici**. Le versioni di questi metodi che non prendono parametri internamente chiamano le versioni statiche passando loro lo stato dell'istanza come parametro. Questo evita la ripetizione di codice pressoché identico all'interno della classe.

Di particolare rilievo è il metodo `guessFollowers()`: internamente, viene chiamato un costruttore della classe che prende in ingresso una lista di Post. Dato che le relazioni tra utenti ("chi segue chi") sono dipendenti dai like e solo da essi, e l'informazione sui like è contenuta nei Post, è possibile ricostruire queste relazioni a partire dalla sola lista dei Post. È quindi sufficiente costruire un'istanza temporanea di MicroBlog a partire dalla lista di Post passati per poter restituire una map contenente tutte le relazioni tra utenti.

Altri metodi di rilievo sono:

- `createPost()`: crea un nuovo Post coi parametri passati al metodo e lo aggiunge alla rete. Verifica se l'utente che ha creato il Post non ha mai postato e, in tal caso, lo aggiunge all'insieme degli utenti registrati nelle varie strutture dati.
- `getUniqueId()`: restituisce un intero sempre diverso, utilizzabile per garantire l'unicità degli id dei Post nella rete.
- `getNumberOfLikedPosts(u, v)`: restituisce il numero di Post di u ai quali v ha messo like (utilizzato dai metodi successivi)
- `likePost()`/`unlikePost()`: utilizzati per mettere o rimuovere like a un post. Essi utilizzano il metodo `getNumberOfLikedPosts()` per verificare quando è necessario aggiornare le relazioni di "follower"
- `getNumberOfFollowers(u)`: restituisce il numero di utenti che seguono u. Utilizzato da `influencers()` per stabilire quali utenti restituire.
- `sortByRelevance()`: permette di utilizzare il social network come motore di ricerca per i Post. Prende in input una lista di stringhe e utilizza una funzione lambda per restituire tutti i Post della rete ordinati decrescentemente per rilevanza, ovvero per numero di parole di ricerca contenute all'interno del proprio testo.

2.3 Funzione di astrazione

Sebbene la gestione ottimizzata dello stato nella classe comporti l'uso di molteplici strutture dati, un'istanza astratta di questa può in realtà essere espressa soltanto tramite un insieme di Post. Come già citato, essi contengono tutte le informazioni necessarie per ricostruire la lista degli utenti e le relazioni tra essi.

Un'alternativa consiste nel definire lo stato astratto mediante due funzioni:

$$f : \text{utente} \mapsto \mathit{set}(\text{utenti}) \mid \forall u, v \text{ utenti } . \ v \in f(u) \iff u \text{ segue } v$$

$$g : \text{utente} \mapsto \mathit{set}(\text{Post}) \mid \forall p \text{ Post}, u \text{ utente } . \ p \in g(u) \iff u \text{ è autore di } p$$

2.4 Dimostrazione di correttezza della classe

Segue la dimostrazione che la classe rispetta l'invariante contenuta nel file `MicroBlog.java`.

Dimostrazione.

- **Caso base**: Il costruttore istanzia le strutture dati come mappe vuote. Pertanto, le condizioni imposte su di esse sono vacuamente vere. Non ci sono post nella rete, quindi anche l'asserzione che la variabile `nextId` sia diversa dall'id di ogni post nella rete è vacuamente vera.

- **Passo induttivo**: Ogni chiamata a `createPost()` aggiunge il nuovo post a `postLookup`, `postRelations` e, se è il primo post da quell'utente, aggiunge anche il suo nome a `followRelations`, mantenendo consistenza tra le strutture.

  Ogni chiamata a questo metodo causa inoltre una chiamata a `getUniqueId()`, che restituisce un intero ogni volta maggiore di 1 del precedente. Gli id restituiti dal metodo non sono mai gli stessi, e dato che gli id dei Post nella rete possono essere solo generati da questo metodo, essi sono unici e diversi dal valore di `nextId` dopo ogni chiamata.

  Ogni chiamata a `addLike()` o `removeLike()` aggiorna se necessario le relazioni tra utenti, eventualmente aggiungendo o rimuovendo follower.

  Tutti gli altri metodi non modificano lo stato, o lo fanno solo mediante chiamate ai sopra citati metodi; pertanto non violano l'invariante.

□

3 Estensione della classe MicroBlog

3.1 Prima proposta: funzionalità di segnalazione dei contenuti

La classe **MicroBlogWithReports** aggiunge il supporto per la segnalazione dei contenuti offensivi all'interno della rete sociale.

Viene definita una *nested class* chiamata **Report** che contiene le informazioni relative a una segnalazione: l'id del post segnalato e il nome dell'utente che ha effettuato la segnalazione. La struttura dati `reports` mappa interi su insiemi di Report, e per ogni chiave contiene l'insieme delle segnalazioni fatte al post con l'id identificato dalla chiave.

La variabile `maxReportCount`, di tipo `byte` poiché assume valori tipicamente piccoli (nell'ordine di 5-10), rappresenta il numero massimo di segnalazioni che possono essere effettuate a un post prima che esso venga censurato. Quando questo limite viene oltrepassato, il metodo `reportContent()` sostituisce il vecchio contenuto del post in questione con la stringa *(deleted)*. Un utente non può segnalare i propri post o segnalare più di una volta lo stesso post.

La classe rispetta il principio di sostituzione in quanto non ci sono metodi *overridden*, e l'unico nuovo metodo introdotto non altera le pre- e post-condizioni.

3.2 Seconda proposta: censura automatica

La classe **MicroBlogWithBadwordFiltering** aggiunge il supporto per la censura automatica dei contenuti offensivi. Al momento dell'istanziazione, viene fornita una lista di parole da censurare all'interno dei post.

Il metodo `createPost()` della superclasse viene *overridden* aggiungendo la funzionalità di censura. Per ogni post creato, viene iterata la lista delle badword e, per ciasciuna di esse, tutte le sue occorrenze all'interno del contenuto del post vengono sostituite dalla stringa *****.

La classe rispetta il principio di sostituzione in quanto l'unico metodo *overridden* `createPost()` non altera le pre- e post-condizioni.