UNIVERSITÀ DI PISA

Dipartimento di Informatica
Corso di Laurea in Informatica

Relazione progetto Java

Programmazione II

Prof. Gianluigi Ferrari

Giacomo Trapani
600124 - Corso A

Anno Accademico 2020/2021

Relazione

Dettagli sul metodo di sviluppo

Il progetto è stato sviluppato con il supporto del text editor *VSCode*, senza tuttavia l'utilizzo di strumenti esterni quali *Maven* per la compilazione o *JUnit* per la creazione dei tests, che invece sono stati scritti a mano. Di conseguenza, l'esecuzione è possibile utilizzando semplicemente i comandi messi a disposizione dal JDK. Ad esempio:

```
cd src
javac *.java
java -ea Main
```

In alternativa si può utilizzare il bash script in allegato:

```
cd src
chmod +x run.sh
./run.sh
```

Dettagli implementativi

Il progetto consiste nello sviluppo di una componente software di supporto alla gestione e l'analisi di un social network denominato **MicroBlog**. Nella rete sociale è consentito inviare messaggi di testo composti da al più 140 caratteri, chiamati **Post**. Agli utenti è consentito seguire i Post (implementato con un meccanismo di Like) e segnalarli.

1 Parte 1

Il tipo di dato **MyPost** rappresenta una collezione immutabile di dati e implementa l'interfaccia **Post**. Un MyPost è definito dai seguenti elementi:

```
1 private final long ID; // identificatore univoco del Post
2 private final String Author; // autore del Post
3 private final String Text; // contenuto del Post
4 private final String Timestamp; // timestamp del Post
```

I parametri vengono implementati come *final* per vietarne la modifica; sono privati ed accessibili attraverso dei getter specifici dichiarati nell'interfaccia **Post**. Sui parametri valgono le condizioni indicate nell'invariante di rappresentazione:

$$ID \geq 0 \ \wedge \ Author \neq null \ \wedge \ Text \neq null \ \wedge \ Timestamp \neq null$$
$$\wedge \ \neg(Author \in \ ”/\backslash A\backslash s*\backslash z/”) \ \wedge \ \neg(Author.isEmpty()) \ \wedge \ \neg(Text.isEmpty()).$$

A garantire l'unicità dell'identificatore univoco *ID* non è la classe MyPost - che rappresenta solamente un contenitore di dati - bensí la classe che lavora sui Post: **MySocialNetwork**.

2 Parte 2

Il tipo di dato **MySocialNetwork** rappresenta una collezione mutabile di dati e implementa l'interfaccia **SocialNetwork**. La progettazione richiede però l'utilizzo di un tipo di dato **Like**: in questa implementazione questo viene considerato un **Post** istanziato con un testo del tipo "$Like : \langle id \rangle$" con id definito come l'identificativo del Post a cui l'autore aggiunge un Like. Da questo punto in poi l'insieme dei Post viene considerato come formato dall'unione degli insiemi **textualPosts** (ossia i Post che non sono Like) e **likePosts** (ossia i Post che sono Like). Su ogni elemento di likePosts vale la seguente condizione:

$$(\forall Post\ p \in likePosts.\ (\exists Post\ q \in textualPosts \mid q.getID() = ReferencedID(p.getText()))).$$

La funzione $ReferencedID(String\ s)$ è banalmente definita come la funzione che estrae la stringa che corrisponde all'identificativo a cui il testo del Post fa riferimento.

Per fare in modo che un Post l sia considerabile un Like valido deve inoltre verificarsi la seguente condizione:

$$(\exists Post\ p \in textualPosts \mid getReferencedID(l.getText()) = p.getID()\ \wedge\ p.getAuthor() \neq l.getAuthor())$$
$$\wedge \neg(\exists Post\ q \in likePosts \mid getReferencedID(q.getText()) = getReferencedID(l.getText())$$
$$\wedge\ q.getAuthor() = l.getAuthor())$$

Nonostante $Post\ p \in textualPosts$ e $Post\ q \in likePosts$ siano implementati dallo stesso tipo di dato, i metodi che lavorano sul contenuto dei Post ignorano q che non si ritiene rilevante.

MySocialNetwork

Il tipo di dato MySocialNetwork è definito dalle seguenti variabili di istanza:

```
// contatore per i Post istanziati all'interno del social network
private long postsNum;
// Follows[a] = insieme degli utenti seguiti da a
protected Map<String, Set<String>> Follows;
// insieme textualPosts
private Set<Post> textualPosts;
// insieme likePosts
private Set<Post> likePosts;
// Followers[a] = insieme degli utenti che seguono a
protected Map<String, Set<String>> Followers;
```

Sulla classe vale il seguente invariante di rappresentazione:

$$Follows \neq null\ \wedge\ textualPosts \neq null\ \wedge\ likePosts \neq null\ \wedge\ Followers \neq null$$
$$\wedge\ postsNum = textualPosts.size() + likePosts.size()$$
$$\wedge\ (\forall String\ i \in [0; Follows.size()].\ i \neq null\ \wedge\ Follows[i] \neq null)$$
$$\wedge\ (\forall Integer\ i \in [0; Followers.size()].\ i \neq null\ \wedge\ Followers[i] \neq null)$$
$$\wedge\ (\forall Post\ p \in textualPosts.\ p \neq null)$$
$$\wedge\ (\forall Post\ p \in likePosts.\ p \neq null)$$
$$\wedge\ (\forall Pair\ \langle k, v \rangle \in Follows.Entry.\ (\forall String\ u \in v.\ (\exists Post\ p \mid p \in textualPosts\ \wedge\ p.getAuthor() = u$$
$$\wedge\ (\exists Post\ q \mid q \in likePosts\ \wedge\ q.getAuthor = k\ \wedge\ getReferencedID(q.getText()) = p.getID()))))$$
$$\wedge\ (\forall Pair\ \langle k, v \rangle \in Followers.Entry.\ (\forall String\ u \in v.\ (\exists Post\ p \mid p \in textualPosts\ \wedge\ p.getAuthor() = k$$
$$\wedge\ (\exists Post\ q \mid q \in likePosts\ \wedge\ q.getAuthor = u\ \wedge\ getReferencedID(q.getText()) = p.getID()))))).$$

L'invariante assicura - oltre all'assenza di parametri di valore *null* - che siano verificate tutte le condizioni necessarie nel meccanismo di follows implementato: un Utente (identificato in modo univoco da un parametro

di tipo **String**) u segue un Utente v se e solo se esiste (almeno) un Like di u a un Post di v; viene mantenuto (per induzione) da due chiamate alla funzione *maintainIR* (una per la mappa dei *Follows*, l'altra per i *Followers*) ogni volta che *this* viene modificato:

```
private static void maintainIR(Map<String, Set<String>> map, String author, String s)
{
    Set<String> tmp = new HashSet<>();
    if (map.get(s) != null)
        tmp = map.get(s);
    if (!tmp.contains(author))
    {
        tmp.add(author);
        map.put(s, tmp);
    }
}
```

La funzione viene chiamata ogni volta che viene creato un Like, aggiunge *author* all'insieme dei valori mappati dalla chiave*s*.

Viene scelto di dividere in due sottoinsiemi i Post presenti in MySocialNetwork per ridurre la complessità del metodo addPost:

```
public Post addPost(String author, String text)
    throws NullPointerException, Post.IllegalPostException
{
    Post p = null;
    try
    {
        p = new MyPost(postsNum, author, text);
    }
    catch (NullPointerException | Post.IllegalPostException e)
    {
        throw e;
    }
    if (SocialNetwork.LikePattern(text))
    {
        try
        {
            return createLike(p);
        }
        catch (Post.IllegalPostException e)
        {
            throw e;
        }
    }
    if (textualPosts.add(p))
    {
        postsNum++;
        SocialNetwork.initializeUser(Follows, Followers, p.getAuthor());
    }
    return p;
}
```

Nel caso in cui il Post che si vuole aggiungere all'insieme dei Post presenti nella rete non segue il formato di un Like questo metodo ha complessità $O(LikePattern(text))$ (ossia un tempo costante); nel caso contrario, la complessità del metodo (al caso pessimo) diventa $O(textualPosts.size()) + O(likePosts.size())$ poiché vengono vietate le operazioni di Like su un proprio Post, uno non presente nel Social, uno a cui lo si è già messo, o su un altro Like - reputate illogiche. In entrambi i casi alla complessità del metodo va aggiunto anche il tempo necessario per mantenere l'invariante della classe.

Si sceglie di mantenere la variabile di istanza *Followers* - corrispondente all'inversa della Map *Follows* - per ridurre al minimo la complessità del metodo influencers:

```java
public List<String> influencers()
{
    List<Entry<String, Set<String>>> list = new LinkedList<>(Followers.entrySet());
    Collections.sort(list, new Comparator<Entry<String, Set<String>>>()
    {
        public int compare(Entry<String, Set<String>> e1, Entry<String, Set<String>> e2)
        { return Integer.compare(e1.getValue().size(), e2.getValue().size()); }
    });
    List<String> res = new LinkedList<>();
    for (Entry<String, Set<String>> l: list) res.add(l.getKey());
    Collections.reverse(res); // now it is in a descending order
    return res;
}
```

Con questa implementazione il metodo *influencers* non deve andare a operare sull'insieme dei Post o sulla Map dei Follows: diventa un metodo di sorting con complessità $O(n \times \log n)$, *con* $n := Followers.entrySet().size()$.

Nel Social Network è presente anche un meccanismo di **tagging**: è possibile menzionare altri utenti presenti nel Social Network all'interno di un generico $Post\ p \mid p \in textualPosts$ con testo contenente una sottostringa del tipo $@Username \mid Username \in Users \ \wedge \ p.getAuthor() \neq Username$, *con* $Users := Follows.keySet()$ (con la funzione $getTagged()$ che banalmente restituisce il nome utente dell'utente menzionato); non viene proibita la creazione di un $Post\ q \mid q.getAuthor() = q.getTagged()$, in casi di questo tipo nel testo non sarà rilevata alcuna menzione. Questo meccanismo è implementato dai metodi che vanno a restituire l'insieme degli utenti menzionati all'interno dei Post; se ne riporta un valido esempio:

```java
private Set<String> detectMention(Post p)
    throws NullPointerException
{
    if (p == null)
        throw new NullPointerException("Post cannot be null.\n");
    Set<String> res = new HashSet<>();
    String delims = "[ ]";
    String[] tokens = p.getText().split(delims);
    for (String t : tokens)
        if (t.startsWith("@"))
            if (!(p.getAuthor().equals(t.substring(1))) && Follows.containsKey(t.substring(1))
    )
                res.add(t.substring(1));
    return res;
}

public Set<String> getMentionedUsers()
{
    Set<String> res = new HashSet<>();
    for (Post p : textualPosts)
        res.addAll(detectMention(p));
    return res;
}
```

3 Parte 3

Il tipo di dato **MyKiddiesSocialNetwork** - collezione mutabile di dati - rappresenta una estensione gerarchica del tipo di dato **MySocialNetwork** in cui viene implementato un meccanismo di **Report** (o **Segnalazione**) e implementa l'interfaccia **KiddiesSocialNetwork**. La progettazione richiede però l'utilizzo di un tipo di dato **Report**: in questa implementazione questo viene considerato un **Post** istanziato con un testo del tipo: "$Segnalazione : \langle id \rangle$" con id definito come l'identificativo del Post che l'autore vuole segnalare. Da questo punto in poi l'insieme dei Post viene considerato come formato dall'unione degli insiemi **textualPosts** (ossia i Post che non sono né Like né Report), **likePosts** (ossia i Post che sono Like) e **reportPosts**. Su ogni elemento

di reportPosts vale la seguente condizione:

$$(\forall Post\ p \in reportPosts.\ (\exists Post\ q \in textualPosts \mid q.getID() = ReferencedID(p.getText()))).$$

Per fare in modo che un Post r sia considerabile un Report valido deve inoltre verificarsi la seguente condizione:

$$\begin{aligned}(\exists Post\ p \in textualPosts \mid getReferencedID(l.getText()) = p.getID()\ \wedge\ \neg(p \in \text{"}\hat{}[A-Za-z]*\text{"})\\ \wedge\ p.getAuthor() \neq l.getAuthor())\\ \wedge\ \neg(\exists Post\ q \in reportPosts \mid getReferencedID(q.getText()) = getReferencedID(l.getText())\\ \wedge\ q.getAuthor() = l.getAuthor()).\end{aligned}$$

3.1 MyKiddiesSocialNetwork

Il tipo di dato MyKiddiesSocialNetwork è definito dalla seguente variabile di istanza (a cui vanno aggiunte quelle di MySocialNetwork):

```
private Set<Post> reportPosts; // insieme reportPosts
```

Sulla classe vale il seguente invariante di rappresentazione:

$$super.IR()\ \wedge\ reportPosts \neq null\ \wedge\ (\forall Post\ p \in reportPosts.\ p \neq null).$$

Poiché la specifica del progetto non prevede un meccanismo di gestione dei post che hanno ricevuto segnalazioni si sceglie di implementare sia un getter per i post segnalati sia per quelli "puliti":

```
public Set<Post> getReportedPosts()
{
    Set<Post> src = new HashSet<>(getPosts());
    Set<Post> res = new HashSet<>();
    for (Post p : getSegnalazioni())
    {
        for (Post q : src)
            if (SocialNetwork.getReferencedID(p.getText()).equals(q.getID()))
            {
                src.remove(q);
                res.add(q);
                break;
            }
    }
    return res;
}

public Set<Post> getFilteredPosts()
{
    Set<Post> res = new HashSet<>(getPosts());
    res.removeAll(getReportedPosts());
    return res;
}
```

Test

Il **file di test** è stato scritto a mano, senza l'utilizzo di framework esterni. Utilizza una stessa funzione per operare sulle due classi, in modo da assicurare che il principio di sostituzione venga rispettato. Per ogni funzione, sono stati controllati anche i relativi **edge cases**, utilizzando scenari limite per lanciare le varie eccezioni. L'esito di ogni test viene stampato nel file "out.txt" su cui viene reindirizzato l'output: in questo si possono vedere le linee in cui le eccezioni sono state correttamente lanciate e - alla fine del file - si troveranno dei commenti accompagnati da una spunta verde ("") nel caso in cui i test siano stati passati con successo.