Relazione Progetto Java PR 2

Venturi Ludovico
Docente: Francesca Levi

UNIPI, Novembre 2019

Indice

1 Scelte progettuali

Nella relazione verranno spiegate le scelte progettuali e implementative che sono state prese.

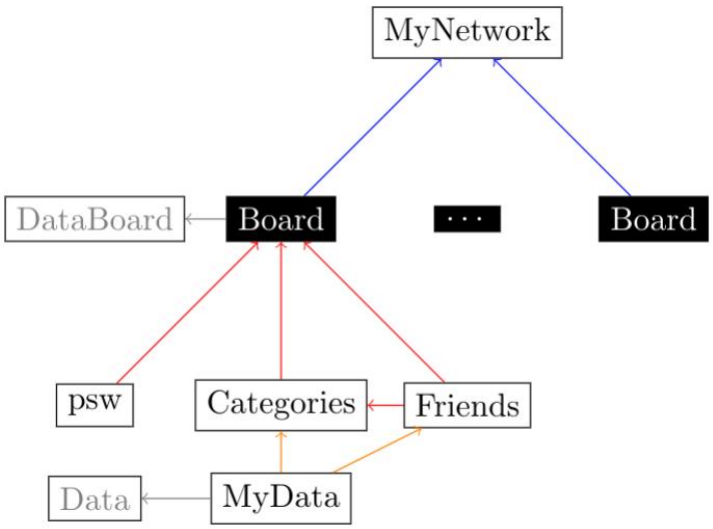

Figura 1: Struttura generale del progetto (comune ad entrambe le implementazioni)

1.1 Data

A grandi linee è stata creata una struttura generale dei dati in grado di adattarsi a varie situazioni.
`Data` viene implementata come *interfaccia*: stabilisce un *contratto* con le sottoclassi che la implementeranno.

OVERVIEW: *Data rappresenta un dato astratto sottoforma di un insieme di 2 attributi e alcune operazioni. È una struttura astratta immutable, di dimensione finita e fissa*

Stabilisce delle caratteristiche comuni a tutte le sottoclassi: metodi che dovranno necessariamente implementare:

```
public void display();
public String getDataTitle();
public String getCategory();
```

1.1.1 MyData

Viene poi implementata una sottoclasse *astratta* `MyData` per soddisfare l'obiettivo iniziale di progettare una struttura versatile.
`MyData` è implementata come classe astratta e tale scelta deriva dalla volontà di attribuire a tutte le classi che discendono da `MyData` delle altre caratteristiche comuni, più *concrete*, ovvero dei metodi già implementati e una struttura implementativa di base:

```
private String dataName;
private String category;
```

Come da specifica la classe `MyData` riporta anche il metodo display, ma astratto: verrà implementato dalle sottoclassi. Da `MyData` discendono i dati veri e propri che saranno salvati in bacheca: nell'esempio per il progetto sono `Testo` e `Audio`, che:

- ridefiniscono `equals()` per permettere la deep equality
- aggiungono un contenuto
- implementano `display()`

1.1.2 Ipotesi

- La bacheca risulta una collezione di oggetti di vario tipo, come riportato nel testo, ma ciò non è direttamente collegato ad ≪`E extends Data`≫. Si è pertanto deciso di riportare anche la classe (magari apparentemente superflua) `MyData` in modo da far trasparire chiaramente la presa di coscienza di ciò. La bacheca gestisce collezioni di dati di tipo E : pertanto ogni dato sottotipo di E, con E definito come ≪`E extends Data`≫, è un dato valido da inserire in una bacheca basata sul tipo E (l' `upcasting` è sempre possibile in queste condizioni); nel progetto si creano bacheche di tipo `MyData` e vengono caricate con dati di tipo `Audio` e `Testo`, entrambi sottotipi di `MyData`
- Non ci sono setter poichè si è ipotizzato che *Data* fosse una struttura *immutable*. (Nel testo viene riportato ≪*i dati possono essere visualizzati dagli amici ma modificati solamente dal proprietario della bacheca*≫: ciò è stato interpretato come: "la modifica consiste nell'aggiunta o la rimozione dei dati, non nella modifica effettiva del contenuto dei dati").

1.2 Board≪E extends Data≫

`Board` rappresenta un contenitore di oggetti generici. È basata sul tipo generico ≪`E extends Data`≫ e funzionalmente è una collezione di dati che possono essere di vario tipo, a patto che siano sottotipi di E (che estende `Data`).

Non ho riportato la specifica di ogni metodo nel codice di `Board(2)≪E extends Data≫` poichè risultava troppo confusionario; l'implementazione ha comunque seguito di pari passo la specifica riportata nell'interfaccia `DataBoard≪E extends Data≫`.

1.2.1 Ipotesi

- non sono ammessi elementi `null`
- non sono ammessi duplicati di alcun genere
- il numero di likes non dipende solamente dal dato ma anche dalla bacheca in cui si trova ⇒ `MyData` o le sue sottoclassi non possiedono il contatore dei like: questo si trova nella bacheca, relativamente ad ogni dato
- la lista ritornata da `getDataCategory` è in sola lettura, avendo precedentemente ipotizzato che Data sia immutable
- la rimozione di una categoria comporta la perdita dei dati in quella categoria e l'associazione per ogni amico di quella categoria
- i likes sono univoci per ogni amico
- `get()` ritorna il dato vero e proprio, non una copia, ma ciò non crea problemi poichè è stato scelto di rendere i dati *immutabili*
- la rimozione della condivisione di una categoria con un amico non comporta la rimozione dei like che quest'ultimo ha assegnato ai vari dati di quella categoria
- ad un dato è associata una e una sola categoria

Nota sugli Iteratori Gli Iteratori sono stati implementati come classi *interne* alla classe `Board`, precisamente come classi di *istanza*. Non sono *static* poichè dipendono dal tipo generico `E` e devono accedere alle variabili di istanza.
Implementano l'interfaccia `Iterator≪E≫`; non implementano il metodo `remove()`, anzi sollevano un' `UnsupportedOperationException` se chiamato.
Quando invocati gli iteratori restituiscono un iteratore di istanza della classe interna (la quale, si ricordi, ha visibilità limitata a `Board`).
Ad esempio:

```
public Iterator≪E≫ getFriendIterator(String friend)
```

quando invocato ritorna un'istanza della classe interna:

`private class FriendIterator implements Iterator≪E≫`

che implementa i metodi `hasNext()` e `next()` e che gestisce al suo interno una struttura dati contenente i dati su cui iterare:

- nel caso di `getFriendIterator` vengono salvati su una struttura di supporto tutti i dati delle categorie condivise con quell'amico
- nel caso di `getIterator` vengono salvati su una struttura di supporto *tutti* i dati in bacheca, dispendioso, ma necessario per poter applicare un ordinamento crescente in base ai like (data la struttura in cui sono memorizzati normalmente). Per gestire questo iteratore è stata creata una nuova classe interna `private class LikeSortedDataIterator implements Iterator≪E≫`.

Gestione delle Password Si è scelto di salvare le password tramite `hash` in array di `byte`[1], il tutto è gestito nella classe `public final class MyPasswordCrypt`.

Eccezioni Sono stati documentati tutti i sollevamenti delle `eccezioni`. In più sono state definiti 3 nuovi tipi di eccezione: `DuplicateLikeException, HiddenCategoryException, WrongPasswordException` tutte sottoclassi di `RuntimeException` e quindi **unchecked**.

1.2.2 Implementazione 1

Per la prima implementazione di `Board` è stata scelta la seguente struttura:

```
private HashMap<String, ArrayList<InternalData<E>>> categories
    private HashMap<String, ArrayList<String>> friends
```

con `InternalData<E>` definito come segue:

```
E data
ArrayList<String> friendsWhoLiked
int likes
```

La scelta di utilizzare `HashMap≪K,V≫` è stata basilare in modo da poter avere associazioni univoche per chiave, e accessi veloci alla struttura. Rimangono però da gestire le chiavi duplicate (non accettabili qui) e/o `null` (ne ammette 1) così come si deve garantire che i valori inseriti nella `HashMap` non siano mai `null`.
`ArrayList≪E≫` è stato scelto poiché di semplice utilizzo come contenitore di una quantità variabile di elementi. Si deve far attenzione a non inserire elementi `null` o duplicati.
`InternalData≪E≫` è un *record* interno alla classe `Board` usato per poter gestire l'associazione DATO-LIKE e valgono le stesse considerazioni per l' `ArrayList` (utilizzato al suo interno) degli altri sopra.

1.2.3 Implementazione 2

Per la seconda implementazione di `Board` è stata scelta la seguente struttura:

```
private HashMap<String, TreeSet<InternalData<E>>> categories
    private HashMap<String, TreeSet<String>> friends
```

con `InternalData<E>` che utilizza `TreeSet<String> friendsWhoLiked` al posto di `ArrayList`, e pertanto valgono le stesso considerazioni precedenti più quelle sotto per il `TreeSet` utilizzato al suo interno.
Per `HashMap` valgono le stesse considerazioni dell'implementazione 1.
`TreeSet` è stato scelto per le sue proprietà molto interessanti: non ammette `null`, non ammette duplicati e salva i dati in ordine crescente. Nonostante questo sono stati comunque effettuati alcuni controlli sui parametri poiché necessari per altre operazione che non riguardavano un `TreeSet`. Da notare che per far sì che il `TreeSet` comparasse a dovere il tipo `InternalData≪E≫` è stato definito un comparatore interno alla classe `Board`. Nell'iteratore invocato in `getIterator()` l'utilizzo del TreeSet ha permesso di evitare l'ordinamento di tutti i dati. Per scorrere ogni `TreeSet` si è abbondantemente fatto uso degli `iteratori`.

[1] https://www.baeldung.com/java-password-hashing

2 Eseguire il codice

Eseguendo il `main` si può testare la struttura già presente che testa tutte le funzionalità. È riportata la prima implementazione di `Board`, per testare la seconda è sufficiente *fare un find and replace di* `" Board"` *con* `" Board2"` *(spazi compresi)*.

2.1 Test ed esempi

Nel `main` sono già presenti un insieme di operazioni che testano il progetto.
Viene creata una `rete` in cui sono presenti 2 bacheche di 2 utenti e, oltre a funzionare, mostra principalmente come vengono gestite le eccezioni. In generale più istruzioni in un singolo blocco `try-catch` indicano che l'ultima sarà quella che genera l'eccezione mentre le precedenti sono eseguite con successo.
Non saranno verificati *tutti* i casi in cui parametri sono null per ovvie ragioni, così come tutte le eccezioni ripetute, quali i controlli che la categoria esista o che l'amico sia presente nella lista amici; per queste ultime, pur se generabili in differenti occasioni, verranno esplicitamente testate una volta ciascuna. Lista di test effettuati nel `main`:

1. password della bacheca $\ll 8$ caratteri
2. get di una bacheca non presente
3. password errata
4. stringa nulla passata (vale per tutti i metodi)
5. aggiunta di una categoria già presente
6. rimozione di una categoria non presente
7. condivisione di una stessa categoria con uno stesso amico
8. condivisione di una categoria non presente nella bacheca
9. rimozione di un amico non presente nella lista amici
10. rimozione di un amico da una categoria cui non ha accesso, anche se presente nella lista amici
11. inserimento di un dato già presente
12. inserimento di un dato la cui categoria non è presente in bacheca
13. get di un dato la cui categoria non è presente
14. get di un dato non presente
15. get di un dato precedentemente inserito in modo corretto ma la cui categoria è stato poi rimossa
16. rimozione di un dato non presente
17. getDataCategory e modifica della lista ritornata
18. amico vuole inserire like ad un dato di una categoria non condivisa con lui
19. amico vuole inserire like ad un dato cui lo ha già messo
20. amico vuole inserire like ad un dato non presente
21. ITERATORI, prove varie
22. elimino una categoria e itero sui dati di tale categoria tramite una amico con cui essa era condivisa