Progetto PR2: prima parte

Alessandro Puccia

1 Traccia

Si richiede di progettare, realizzare e documentare la collezione `SecureDataContainer<E>`. `SecureDataContainer<E>` è un contenitore di oggetti di tipo E. Intuitivamente la collezione si comporta come un Data Storage per la memorizzazione e condivisione di dati (rappresentati nella simulazione da oggetti di tipo E). La collezione deve garantire un meccanismo di sicurezza dei dati fornendo un proprio meccanismo di gestione delle identità degli utenti. Inoltre, la collezione deve fornire un meccanismo di controllo degli accessi che permette al proprietario del dato di eseguire una restrizione selettiva dell'accesso ai suoi dati inseriti nella collezione. Alcuni utenti possono essere autorizzati dal proprietario ad accedere ai dati, mentre altri non possono accedervi senza autorizzazione.

2 Scelte progettuali

La principale scelta progettuale che ho fatto è stata la distinzione tra utente proprietario del dato e utenti autorizzati ad accedere al dato. In tal senso ho definito una classe di supporto `Dato<E>` che indicherà, oltre all'oggetto di tipo E, l'utente proprietario e l'insieme degli utenti autorizzati ad accedervi. Ho definito inoltre un'altra classe `Utente` che conterrà l'id e la password per semplificare le strutture. Per quanto riguarda i metodi ho implementato le seguenti scelte:

- `public void createUser(String user, String passw)`
  Aggiungo user se non era già presente.
- `public void removeUser(String user, String passw)`
  Chiamo il metodo `remove` per tutti gli elementi dell'utente, l'utente viene rimosso dalla struttura.
- `public boolean put(String user, String password, E data)`
  - Se user vuole inserire un oggetto E passandomi un riferimento che non è presente nella collezione, esso verrà aggiunto e restituisce al chiamante true.
  - Se user vuole inserire un oggetto E passandomi un riferimento che è presente nella collezione ma posseduto da un altro utente, esso verrà aggiunto (modo deep) e restituisce al chiamante true.
  - Se user vuole inserire un oggetto E passandomi un riferimento che è presente nella collezione ma posseduto da user, esso non verrà aggiunto e restituisce al chiamante false.
- `public void getSize(String user, String password)`
  Restituisce il numero di oggetti di tipo E posseduti da user, non vengono conteggiati elementi posseduti da altri utenti e dati in condivisione a user.
- `public E get(String user, String password, E data)`
  Restituisco una copia shallow di data se è presente.
- `public void copy(String Owner, String passw, E data)`
  Crea una nuova copia di data (modo deep).

- `public void share(String user, String password, E data)`
  Restituisco una copia (modo shallow) di data se è presente, in questo modo qualsiasi modifica fatta da other si ripercuote immediatamente sul dato (non c'è necessita di ricondividere il dato da parte di other se lo modifica).

- `public E remove(String user, String password, E data)`
  Se user è proprietario di data, esso verrà cancellato a tutti gli utenti a cui era stato condiviso, altrimenti viene cancellato solo ad user e il suo id verrà tolto dall'insieme degli autorizzati.

- `public Iterator<E>(String user, String password)`
  Implementato utilizzando una inner class `EGenerator`.

La copia deep viene gestita in tutti i casi in cui viene utilizzata attraverso `Serializable`.

3 Implementazioni

NB: per il corretto funzionamento delle implementazioni si assume che l'oggetto di tipo E abbia fatto override del metodo `equals()` in modo opportuno e abbia implementato la classe `Serializable`.

3.1 `HashTable<K, V>`

Per questa implementazione ho utilizzato un `ArrayList<Utente>` per poter contenere tutti gli utenti registrati e una `HashTable<Utente, ArrayList<Dato<E>>` per contenere i dati. In particolar modo è stato necessario fare override dei metodi `hashCode()` e `equals()`, della classe `Utente`, per utilizzare in maniera corretta la tabella hash. La funzione hash utilizzata è stata: `37 * user.hashCode()`, in cui user è la stringa che rappresenta l'id dell'utente.

3.2 `TreeMap<K, V>`

Per questa implementazione ho utilizzato solamente una `TreeMap<Utente, ArrayList<Dato<E>>` in cui è stata necessario fare override del metodo `equals()` e `compareTo()`, della classe `Utente`, per poter navigare l'albero. Per poter verificare se un utente user è registrato, ho utilizzato il metodo `floorKey(K key)` implementato da `TreeMap<K, V>`.