UNIVERSITÀ DI PISA

Nome*: Cristiana*
Cognome: *Angiuoni*
Matricola: *546144*

RELAZIONE PROGETTO JAVA 2018/2019

“Sistema per la gestione di utenti e di dati ad essi associati”

1. Specifica e analisi del problema:

- Il progetto si pone come obiettivo quello di realizzare la collezione *SecureDataContainer<E>,* ovvero un "contenitore" (*DataContainer*) di O*wner*, che dopo essersi correttamente registrati *(Secure),* possono inserire i propri dati e fare alcune operazioni su essi.
  Lo *scopo* è quello di gestire:
  - **utenti** : caratterizzati da Id e Password.
  - **insieme di dati** : relativi a un preciso utente propretario (O*wner*).

2. Implementazione e strutture dati:

- **Registrazione e verifca dell'identità**: chiunque voglia registrarsi nel sistema deve scegliere un *Id* e una *password* (non nulli*);* non è possibile scegliere un Id già usato da qualcun'altro (altrimenti viene lanciata *NameAlreadyExistsException(), checked exception* gestita con *try-catch),* mentre per le password non ci sono vincoli di duplicazione.

  Un utente viene, dunque, inserito nel sistema*,* diventando da quel momento in poi *Owner* dei propri dati e solo su essi potrà operare. Ogni volta che accede, Owner, deve inserire le proprie credenziali, in modo che Id e passw corrispondano a quelle della precedente registrazione (il metodo privato **VerifiedIdentity** autorizza l'accesso o lo rifiuta (lanciando *FailedAuthenticationException(), checked exception* gestita con *try-catch*).
- **Operazioni sui dati**: se l'identificazione è avvenuta con successo Owner può *aggiungere* un dato (se non è già presente, altirmenti lancia *DuplicateDataException(), checked exception gestita con try-catch*); può *copiarlo*, incrementando il relativo numero di occorrenze solo se ne è proprietario, altrimenti viene lanciata *NotFoundDataException()*, *rimuoverlo*, decrementando il numero di occorrenze se questo è maggiore di 1, altrimenti eliminando il dato stesso, *ottenerne una copia* o *condividere* un dato con un *Other*, che, solo dopo essere stato correttamente identificato con **VerifiedIdentity** diventa proprietario a tutti gli effetti di una copia di quel dato, ovvero Owner aggiunge il suo dato tra quelli di Other, se quest'ultimo non lo possedeva già (altrimenti viene lanciata *DuplicateDataException(), checked exception gestita con try-catch*). Inoltre Owner può *contare* i propri dati e *iterare* sull'insieme dei dati, ottenendo per ciascuno anche il relativo numero di occorrenze.

2.1 Prima implementazione: *MyHashSecureDataContainer<E>*

- struttura dati: **HashTable** di Java, dove ogni indirizzo hash è calcolato a partire dall'Id (*key,* univoca visto che non esistono Id duplicati), il cui *value* associato è la password e il Vector<Data<E>> di dati e occorrenze.

- La funzione hash è **h: {** (String) **Id } ---> {** (UserInfo<E>) **informazioni relative a Id }**.
- La classe *UserInfo<E>* definisce un nuovo tipo di dato, non primitivo, costituito dalla coppia: *<(String)* ***password,*** *Vector<Data<E>* ***D****>>,* in cui avviengono le effettive operazioni sui dati. Le operazioni effettive sui dati avvengono in questa classe, tramite la chiamata di metodi da parte della classe *MyVectSecureDataContainer<E>*, in modo da *modularizzare* e favorire *modificabilità* e *riuso* del codice.
- La classe *Data<E>* definisce un nuovo tipo di dato, non primitivo in Java, costituito dalla coppia: *<(<E>)* ***dato,*** *(int)* ***numero occorrenze,*** *(int)* ***contaDati****>.*

2.2 Seconda implementazione: *MyVectSecureDataContainer<E>*

- struttura dati: **Vector** di Java
- *Vector<Utente<E>>* ***U:*** ogni locazione rappresenta un utente diverso.
- *La classe Utente<E>* è caratterizzata dalle proprie credenziali di accesso, velocemente verificabili, una volta trovato l'Id e il *Vector<Data<E>>* associato, di cui risulta proprietario.
- La classe *Data<E>* definisce un nuovo tipo di dato, non primitivo in Java, costituito dalla coppia: *<(<E>)* ***dato,*** *(int)* ***numero occorrenze del dato,*** *(int)* ***contaDati****>.* *(COME PRIMA)*

2.3 Differenze computazionali:

La prima implementazione, con *Hashtable*, ottimizza le **ricerche puntuali** degli utenti per Id (O(1)). Con un unico accesso, conoscendo la chiave, è possibile sia verificare che la password sia corretta, sia accedere ai dati. Inoltre l'**inserimento** (non ordinato) di un nuovo utente è efficiente (O(1)), svantaggiando ad esempio le **ricerche per intervalli** o la **ricerca di un dato** (che è lineare in *n* in ogni caso, avendo utilizzato un vettore di *n* dati).

Proprio per capire la differenza computazionale, la seconda implementazione, con Vector<Utente<E>>, in cui, in ogni locazione, c'è *Vector<Data<E>> per i dati* oltre all'Id e alla password di ognuno, è in generale sicuramente **meno efficiente** della prima per quanto riguarda le ricerche puntuali, non essendo ad accesso diretto; anche se per **ricerche massive**, scansioni lineari di tutti gli utenti, utili per esempio per scopi statistici, il costo computazionale è O(n), necessario in ogni caso per leggere tutti gli utenti, indipendentemente dalla struttura dati scelta.

3. Testing del programma:

HashMainTest e VectMainTest: classi in cui testo ogni metodo facendo in modo che ogni operazione avvenga con successo:

- **Inserisco 3 utenti: <User 1,Passw 1>, <User 2,Passw 2>, <User 3,Passw 3>.**
- **Aggiungo {d1,d2,d3} in User 1 e {d1,d4} in User 3**, quindi ci sono 3 elementi in User 1, 0 in User2 e 2 elementi in User 3.
- **User 1 condivide il suo dato 'd3' con User 2**; User 2 non ce l'aveva prima della share, quindi viene aggiunto alla collezione di User 2, che ora possiede un dato {d3}.
- **Ora ci sono 3 elementi in User 1, 1 in User 2 e 3 elementi in User 3.**
- **User 1 ottiene la copia il suo dato 'd1'** , e restituisce tale copia, non incrementando l'occorrenza. Invece chiedendo la copia di 'd4', non presente in User 1, **retituisce null.**
- **User 1 copia 4 volte il suo dato 'd2'**, aumentando le occorrenze di 'd2' a 5 nella sua collezione.
- **User 1 condivide il suo dato 'd3' con User 3**; User 3 non ce l'aveva prima della share quindi viene aggiunto alla collezione di User 3, che ora possiede 'd3'.

- **User 3 rimuove ‘d1’** che è presente tra i suoi dati, eliminandolo, perché l’occorrenza era 1 .
- **User 1 rimuove il suo dato ‘d2’** che è presente 5 volte, decrementando le occorrenze, che diventano 4. Infine, le iterazioni sui dati di **User 1: <d1,1> ,<d2,4>, <d3,1>**, **User 2 :<d3,1>** e **User 3 <d4,1>,<d3,1>**.

  ExcHashMainTest e ExcVectMainTest: classi in cui testo fanno emergere le situazioni limite in cui vengono lanciate tutte le eccezioni (, ognuna con il proprio messaggio di errore :
- **“Nome già esistente!”** per *NameAlreadyExistsException()* , **“Hai inserito un parametro nullo!”** per *NullPointerException()* ,**“Non sei registrato!”** per *FailedAuthenticationException()* , **“Il dato è già presente!”** per *DuplicateDataException()*, **“Dato non trovato!”** *NotFoundDataException()*.