RELAZIONE PRIMO PROGETTO – PROGRAMMAZIONE 2 [JAVA]
Matteo Montalbetti

La consegna del progetto poneva il seguente problema: "*[…] si richiede di progettare, realizzare e documentare la collezione SecureFileContainer. SecureFileContainer è un contenitore di oggetti di tipo E. Intuitivamente la collezione si comporta come una specie File Storage per la memorizzazione e condivisione di file. La collezione deve garantire un meccanismo di sicurezza dei file fornendo un proprio meccanismo di gestione delle identità degli utenti. Ogni file ha un proprietario che ha diritto a leggere, scrivere e fare una copia. La collezione deve, inoltre, fornire un meccanismo di controllo degli accessi che permette al proprietario del file di eseguire una restrizione selettiva dell'accesso ai suoi file inseriti nella collezione. Alcuni utenti possono essere autorizzati dal proprietario ad accedere ai suoi file (in solo lettura o anche scrittura) mentre altri non possono accedervi senza autorizzazione. Ma l'utente deve accettare la condivisione previa autenticazione […]*"

Ho deciso di interpretare il meccanismo di *SecureFileContainer* come una specie di funzione matematica, che associa ad ogni utente, una collezione di file {[id, psw] → [file0, … , fileN]}. Un utente è identificato dai suoi dati di accesso, i quali dovranno essere utilizzati per eseguire le varie operazioni sulle proprie collezioni.

Analizziamo le diverse classi che ho utilizzato per la progettazione del *SecureFileContainer*:

- **User →** la classe *User* rappresenta l'astrazione di un certo utente che utilizza la collezione. Ogni utente ha le proprie credenziali di accesso, le quali dovranno essere indicate dallo stesso per autorizzare certe operazioni sui propri file. Non possono esistere due clienti con uguali *id* all'interno della collezione (cosí come nei sistemi di registrazione online).
- **File →** la classe *File* rappresenta l'astrazione di un certo *File* di un certo *User*. Un *File* viene inteso come un oggetto di due attributi: il file vero e proprio (di tipo generico *E*) e i diritti di accesso che l'utente possiede su quel file. Ad ogni utente é infatti associata una collezione, che elenca e memorizza tutti i file (con i rispettivi diritti) dell'utente stesso. Ho deciso di interpretare i diritti di accesso come una stringa di valore "w" o "r" (rispettivamente scrittura e lettura); "w", oltre a rappresentare i diritti di accesso in scrittura, denotano anche la proprietà del file stesso. Un utente aventi diritti "w" diventa difatti proprietario del file (o della propria copia del file).
- **SecureFileContainer →** la classe *SecureFileContainer* é l'interfaccia principale, così come proposta dal progetto. Ho aggiunto all'interno di essa alcune funzioni, quali:
    1. *userSearch(String id)* → cerca l'utente avente come *id* il parametro passato alla funzione e ne torna la posizione. In caso non dovesse trovarlo, la funzione torna *-1*.
    2. *fileSearch(E file, ArrayList<E> fileColl)* → cerca all'interno di una collezione di file (file inteso come dato di tipo *E*, **non** oggetto *File*) un certo file. Anche in questo caso torna la posizione oppure *-1* in caso di assenza del file.
    3. *onlyData(ArrayList<File<E>> array)* → ho spiegato precedentemente il modo in cui ho deciso di interpretare il concetto di *File* all'interno della mia soluzione. Naturalmente questo ha portato ad avere, associato a ciascun *User*, non un semplice *ArrayList* di tipo *E*, bensì un *ArrayList* di tipo *File<E>*, essendo *File* un oggetto. Ho quindi creato una funzione che, preso in input un *ArrayList* di tipo *File<E>*, estrapola da ogni oggetto l'attributo dato (il dato effettivo di tipo *E*) e crea un nuovo *ArrayList* contenente appunto i dati estrapolati, di tipo E.
- **MySecureFileContainer →** la classe *MySecureFileContainer* rappresenta la prima implementazione. Ho deciso di utilizzare come struttura dati due *ArrayList*. Il primo, di tipo *User*, contenente i vari utenti della collezione; il secondo é invece di tipo *ArrayList<File<E>>*: contiene le diverse collezioni di file dei diversi utenti. Il meccanismo é basato sulla corrispondenza degli indici (nei due *ArrayList* principali) di *User* e propria collezione di *File*. (p*er quanto riguarda A.F / I.R / Overview / Specifiche più approfondite consultare i commenti all'interno del codice.)*
- **MySecondSecureFileContainer →** la classe *MySecondSecureFileContainer* rappresenta la seconda implementazione. In questo caso ho invece deciso di utilizzare una *HashMap*, ovvero una tabella chiave

valore. Le chiavi sono rappresentate dagli User, i quali hanno associati (come valore) un ArrayList di tipo File<E>. Rispetto al meccanismo precedente, reputo questa soluzione più corretta e precisa; sicuramente più funzionale. (p*er quanto riguarda A.F / I.R / Overview / Specifiche più approfondite consultare i commenti all'interno del codice.)*

- **testClass** → la classe *testClass* contiene i vari test che ho progettato. Non ci sono particolari istruzioni per eseguirli, basta semplicemente lanciare la classe come *Java Application* e leggere il risultato in output (nella cartella é presente un file di testo *TestOutput.txt* contenente l'output risultato dai miei test). Ho deciso di testare entrambe le implementazioni con uguali comandi. Partendo dalla prima, e successivamente con la seconda, ho testato tutte le funzioni implementate, sia in casi ottimali sia in presenza di errori vari, testando così anche l'efficacia e la correttezza della copertura data dalle eccezioni.
- **ExistingUserException** → la classe *ExistingUserException* é stata creata per lanciare un'eccezione *checked*. Un'eccezione di questo tipo viene lanciata quando si tenta di creare un *User* associandogli un *id* già registrato nella collezione.
- **FileNotFoundException** → la classe *FileNotFoundException* é stata creata per lanciare un'eccezione *checked*. Un'eccezione di questo tipo viene lanciata quando si tenta di operare su un certo *File*, il quale non é però presente all'interno della collezione.
- **RightsErrorException** → la classe *RightsErrorException* é stata creata per lanciare un'eccezione *checked*. Un'eccezione di questo tipo viene lanciata quando si tenta di fare un'operazione su un certo *File* senza avere però i permessi adeguati (ad esempio condividere un file di cui non si é proprietari, ovvero di cui non si hanno diritti in "w")
- **ShareWithYouException** → la classe *ShareWithYouException* é stata creata per lanciare un'eccezione *checked*. Un'eccezione di questo tipo viene lanciata quando si tenta di condividere un *File* con se stessi. Per fare un'operazione di questo tipo esiste l'apposita funzione *copy.*
- **UserNotFoundException** → la classe *UserNotFoundException* é stata creata per lanciare un'eccezione *checked*. Un'eccezione di questo tipo viene lanciata quando si tenta interagire, in qualche modo o con qualche operazione, con un *User* che però non é presente all'interno della collezione.
- **WrongPswException** → la classe *WrongPswException* é stata creata per lanciare un'eccezione *checked*. Un'eccezione di questo tipo viene lanciata quando si tenta di accedere con le proprie credenziali inserendo una password non corretta.

Parlare di come ogni metodo é stato implementato, per entrambe le implementazioni, porterebbe via troppo spazio. Indicherò, quindi, solo alcuni punti, i quali rappresentano alcune scelte o precisazioni ritenute importanti e/o particolari.

1. **Gestione dei diritti di accesso** → come scritto sopra, nella spiegazione della classe *File,* ho deciso di implementare il file stesso come un oggetto, facendo cosi in modo che i diritti di accesso diventino parte di esso. I diritti sono rappresentati come una *String.* Possono valere solo "w" e "r". Altri valori non saranno ammessi e puniti con una *IllegalArgumentException*. Se un utente possiede, nella sua collezione, un *File* con diritti di accesso "w", esso potrà essere definito il proprietario del file stesso. Naturalmente quando i diritti di accesso sono in scrittura, non ci sono restrizioni sulle operazioni da fare. Quando invece i diritti di accesso sono in lettura ho ritenuto opportuno non permettere la condivisione del file, in nessuna delle due modalità. Per le altre operazioni ho deciso di non mettere vincoli; questo perché ogni utente agisce semplicemente su file (e/o copie di file) contenuti all'interno della propria collezione, e non potrà in nessun modo interferire nelle collezioni altrui. Ultima precisazione: quando viene fatta una *copy* di un certo *File,* é vero che non sono presenti vincoli riguardo i diritti, ma é anche vero che la copia avrà uguali diritti dell'originale. (i diritti non potranno in nessun modo passare da lettura a scrittura)
   N.B -- un File inserito con put sarà sempre fornito di diritti in scrittura.
2. **Iteratore senza remove** → per risolvere questo problema, ho deciso di creare una classe *private* interna, la quale implementa l'interfaccia *Iterator.* Ho così astratto un mio iteratore "personalizzato" ridefinendo il metodo *remove,* in modo tale da lanciare una *UnsupportedOperationException* ogni qual volta si tenta di utilizzare questa funzione*. (viene chiesto di implementare un iteratore **senza remove**)*
3. **La condivisione** → ho deciso di implementare la condivisione di un file come un copiare il file stesso all'interno della collezione associata all'utente con cui si vuole condividere.

TEST:
Per quanto riguarda i test, come già spiegato con la presentazione della classe *testClass,* non ci sono particolari indicazioni. Occorre semplicemente lanciare la classe come *Java Application* e verificare l'output. (*TestOutput.txt)*