Relazione SecureDataContainer

Nella mia scelta progettuale un contenitore sicuro di dati può essere immaginato come un insieme di coppie, in cui nel primo membro si trovano ID e password dell'utente, mentre il secondo membro contiene nuovamente un insieme di coppie, che rappresentano la lista di dati di proprietà dell'utente, con la lista degli utenti autorizzati ad accedervi.
Questo permette di separare la figura del *proprietario* dei dati, da chi invece ne ha accesso solamente in lettura.
Questa scelta è stata fatta orientando la collezione sulla sicurezza.
Il titolare di un dato può infatti condividerlo con un gruppo di utenti G (anche vuoto) ed un utente di questo gruppo è autorizzato ad estrarne una copia – da inserire nella propria collezione o da modificare – e visualizzarne il contenuto. Non può però in alcun modo modificare o rimuovere il dato originale.
Coerentemente, ogni metodo produttore restituisce una deep copy del dato, in modo che questo possa essere modificato solo dal suo creatore e soltanto tramite il metodo *Modify* della classe stessa.
Allo stesso modo viene intesa **privata** l'informazione in merito alla quantità di dati di un utente; si scinde quindi quest'informazione, dalla quantità di dati a cui un utente ha accesso e per la quale esiste un metodo apposito (*PrintAuthorizedData*).

In entrambe le implementazioni ho scelto di utilizzare due classi ausiliarie: *User* per gli utenti, *DataInformation* per rappresentare i dati.
Questo permette di tenere facilmente traccia degli utenti con cui è stato condiviso il dato.

In merito alla copia di oggetti generici, la scelta che ho fatto, data l'intenzione di creare copie reali, è quella di vincolare la struttura ad utilizzare solo oggetti che estendano la classe ***MyCloneable.***
Sarà quindi interesse dell'utente che vuole utilizzare la classe sovrascrivere il metodo Clone in modo che vengano restituite copie reali dell'oggetto.

Il metodo *put* inserisce una deep copy dell'oggetto passato come parametro.
Si avrebbe altrimenti una situazione di aliasing, sfruttando la quale sarebbe possibile modificare la collezione dall'esterno.
Il metodo *get* può essere invocato – nonostante il nome fuorviante del parametro – da chiunque abbia accesso a quel dato.
Il parametro Owner, infatti, non identifica il proprietario del dato, ma l'ID di un utente che ne abbia anche solo accesso in lettura.
Il metodo *copy*, allo stesso modo, copia un dato a cui l'utente Owner – non necessariamente proprietario del dato – è autorizzato ad accedere, inserendone una copia nella sua collezione personale.
La copia di questo dato è vista come un nuovo oggetto, uguale solo nel contenuto.
Questa scelta si adatta bene anche nel caso in cui *Owner==Titolare* del dato.
Infatti un utente potrebbe voler effettuare una copia di backup del suo dato o crearne una copia da modificare. Non trovo insensato assumere che questi voglia tenere private le modifiche del dato.
Il metodo *getIterator* restituisce un iteratore sui dati di proprietà dell'utente, escludendo quelli condivisi con egli da altri utenti.
Per accedere ai dati condivisi con l'utente, è possibile utilizzare il metodo *PrintAuthorizedData*, che restituisce un set contenente copie dei dati condivisi con questi.
Il metodo *AuthorizedSize* restituisce il numero di utenti con cui un dato è stato condiviso.
Il metodo *Modify* modifica il dato *OldData*, presente nella collezione, con una copia del dato *NewData* passato come parametro.

In merito alla funzione di cifratura, ho deciso di seguire la stessa idea del meccanismo di creazione di copie.

La classe ***MyCloneable*** conterrà infatti due ulteriori metodi - *encrypt()* e *decrypt()* - che si limitano a restituire l'oggetto sul quale sono state invocate.
Questo è dovuto sia ad una maggiore libertà di scelta di chi utilizza il dato, che può sovrascrivere i metodi con la funzione di cifratura più opportuna, sia ad un difficile adattamento del meccanismo rispetto i generici.
Un metodo che prende come parametro un oggetto generico di tipo E, e ne restituisca un nuovo oggetto di tipo E cifrato, senza fare assunzioni su E, non è semplice.
Quello che ho fatto è sfruttare l'unica informazione che avevo sul tipo E: è una sottoclasse di ***MyCloneable***.
Il metodo *Encrypt()* consente sia di cifrare che di decifrare, in funzione del secondo parametro passato, l'oggetto corrente.
Il metodo si limita, infatti, a richiamare il metodo *encrypt()* o *decrypt()* dell'oggetto.

La classe *User* è formata da due stringhe: ID e password, e pochi metodi per operare su questi.
Tra questi non è presente un metodo che esponga la password, ritenuto insicuro, ma piuttosto un metodo che effettua un controllo d'identità con le stringhe passate come parametri.
È presente inoltre un metodo che verifichi la consistenza delle variabili d'istanza per garantire l'ipotesi di invarianza.
La classe *DataInformation* offre, tra gli altri, un metodo *AddLicense* che permette di aggiungere un utente alla lista degli utenti autorizzati a quel dato.
Entrambe le classi, inoltre, contengono una sovrascrittura del metodo *equals()*.
Questo si è rivelato necessario per poter utilizzare, senza comportamenti inaspettati, i metodi *Contains()* e *IndexOf()* delle strutture di supporto.

La batteria di test è formata da due metodi richiamati, una volta per ogni implementazione, nel main.
Questi si occupano di effettuare, rispettivamente, test legittimi il primo e test estremi il secondo e stamparne i risultati.
Il secondo test non stamperà che una sfilza di eccezioni.
Per i test vengono utilizzati oggetti di tipo **MyType**: classe di oggetti che rappresentano un qualsiasi altro tipo di dato sul quale si vuole operare, con in più la definizione del metodo *Clone()*.
La classe **MyType** sovrascrive anche il metodo *toString()* in modo da restituire una stringa rappresentante le variabili d'istanza dell'oggetto corrente.
Questo è dovuto al fatto che il confronto tra oggetti di tipo *DataInformation* sfrutta il metodo *toString()*. Qualora questo non venisse sovrascritto, dunque, non si garantisce un normale comportamento della struttura.
La classe di test contiene un terzo metodo, commentato, che restituisce il tempo e la memoria occupata per fare operazioni su grandi quantità di dati, con oggetti complessi (**MyComplexType**).
Questo test effettua l'inserimento di 10k utenti e, per ognuno, 30 dati, per un totale di 300000 dati.
Effettua poi la condivisione della prima metà dei dati di ognuno dei primi 5k utenti con la restante metà, finendo poi per rimuovere 500 utenti in maniera pseudocasuale.
Effettuando il test sulla mia macchina risulta una differenza di ≅ 18 sec. e ≅ 60 MB a favore della seconda implementazione, che vede utilizzare un *Hashmap* come struttura di supporto, a dispetto dei *due vettori* utilizzati nella prima.
Da notare la snellezza dell'IR della seconda implementazione, rispetto la prima, indice di una struttura meno soggetta a controlli e, quindi, più sicura.

Istruzioni per la compilazione:
cd SecureDataContainer
*javac *.java MyExceptions/*.java*
java TestBattery
Marco Antonio Corallo, 531466