Università di Pisa
Corso di Programmazione II (273AA)

Progetto Java - Relazione finale

Alessandro Antonelli
(matricola 507264, corso A)

Sessione estiva-autunnale A.A. 2018/2019
Appello settembre 2019

1 Come compilare il codice ed eseguire i test

Per ottenere l'eseguibile, devono essere compilati tutti i file sorgente contenuti nella cartella `src` e sottocartelle. Il codice è organizzato in un package `pr2` (che contene l'interfaccia, una classe ausiliare e la batteria di test) con i tre subpackage `exceptions` (che contiene le eccezioni personalizzate), `impl1` e `impl2` (che contengono le due implementazioni). Il codice è stato sviluppato tramite l'IDE Eclipse e testato su JVM versione 11 in Ubuntu 18.04 64 bit.

La batteria di test è contenuta nel `main` della classe `ClasseTest`; non sono richiesti argomenti o altro tipo di input e l'esito sarà stampato sullo standard output. La batteria è costituita da tre test: il primo istanzia il tipo `E` con `Integer` (un tipo predefinito immutabile), il secondo con `Date` (un tipo predefinito mutabile), il terzo con `MioTipoRecord` (un tipo user-defined apposito), che vengono eseguiti su ciascuna delle due implementazioni per un totale di sei test.

Restrizioni al tipo generico

Nell'uso della collezione (anche per eventuali test aggiuntivi) è richiesto che il tipo generico `E`, che rappresenta il contenuto dei file, sia istanziato esclusivamente con classi che 1) implementino l'interfaccia `Serializable`; 2) ridefiniscano in modo significativo il metodo `equals`, con *override* di quello predefinito. In caso contrario non è garantito il funzionamento.

La prima restrizione è necessaria per la *deep copy* sui generici (come spiegato in dettaglio più avanti), la seconda per poter confrontare l'argomento dei metodi con i file della collezione, e quindi capire su quale di essi si intende operare.

2 Principali scelte progettuali nella specifica

Ammissibilità di file duplicati

È stato scelto di ammettere la presenza di file duplicati nella collezione: infatti i metodi `put`/`share` non sollevano alcuna eccezione se il file è già presente. La scelta è stata pressoché obbligata dalla presenza del metodo `copy` nella consegna fornita dai docenti, che implicitamente lo richiede.

La conseguenza è che non c'è un modo univoco di riferire un file della collezione: infatti un oggetto di tipo `E` potrebbe eguagliare più file della stessa collezione (se ci sono duplicati), e introdurre degli identificatori univoci avrebbe costretto a violare la sintassi dei metodi indicata nella consegna (che utilizzano il dato `E` come identificatore).

Pertanto in tutti i metodi si è scelto si adottare la seguente semantica: il metodo opera su un solo file scelto arbitrariamente tra quelli che eguagliano il file passato per argomento (secondo il metodo `equals`); non c'è modo di sceglierlo né si possono fare assunzioni su quale verrà scelto tra i tanti.

Memorizzazione e condivisione dei file tra utenti

Nonostante un file possa essere accessibile a più utenti, è stato deciso di mantenere in memoria una sola copia del contenuto, di cui gli utenti autorizzati condividono un riferimento (*shallow copy*). La `share` si limita a creare un nuovo puntatore al dato, viceversa la `remove` cancella quello dell'utente che l'ha invocata (non provoca, invece, l'eliminazione fisica del file: questa viene eseguita solo alla rimozione dell'ultimo puntatore).

Questa scelta è stata adottata perché semplifica la collaborazione tra utenti: essendo il dato unico, le modifiche diventano immediatamente visibili a tutti gli utenti con accesso al file; al contrario, memorizzando copie diverse per ogni utente (*deep copy*), ad ogni modifica ci sarebbe stato bisogno di costose operazioni di sincronizzazione tra di esse.

Modalità di modifica del contenuto

Per eseguire modifiche al contenuto dei file si è scelto di non inserire dei metodi appositi (che non sono previsti nella consegna), ma di limitarsi a fornire dei cammini d'accesso al contenuto dei file, che permettano agli utenti autorizzati (e solo ad essi) di eseguirvi modifiche. La realizzazione degli opportuni metodi setter per operare sul contenuto viene delegata all'implementatore del tipo di dato `E`.

Cammini d'accesso di sola lettura tramite *deep copy*

Dal momento che tutti i collaboratori dispongono di un puntatore all'(unico) contenuto del file, è stato necessario prendere accorgimenti per differenziare l'accesso in sola lettura da quello in scrittura, ossia per evitare che gli utenti con accesso in sola lettura potessero usare il puntatore per eseguire modifiche indebite.

A tale scopo si è deciso di agire solamente su come vengono restituiti all'utente i riferimenti: infatti nella specifica si impone che il metodo `get` e l'`iteratore` restituiscano un riferimento "non modificabile" quando l'utente ha accesso al file in sola lettura (ossia eventuali modifiche al riferimento restituito non devono propagarsi al dato memorizzato nella collezione), mentre se l'utente ha accesso anche in scrittura è esplicitamente richiesto che il riferimento sia "modificabile" e che i cambiamenti si propaghino alla collezione. In sintesi, la differenza tra l'accesso in sola lettura e quello in scrittura è che i cammini di accesso "autentici" al dato (utilizzabili per modificarne il contenuto) sono forniti solo nel secondo caso.

A livello di implementazione, queste imposizioni sono state rispettate restituendo per gli accessi in sola lettura una *deep copy* del contenuto (ossia eseguendo *copy-out*) e per gli accessi in scrittura una *shallow copy* (cioè evitando deliberatamente il *copy-out*). Le note limitazioni sulla clonazione dei generici in Java sono state superate tramite l'uso della serializzazione (meccanismo che è stato preferito rispetto all'utilizzo di `clone`, perché implementare l'interfaccia marker `Cloneable` non avrebbe garantito la reale presenza del metodo nella classe `E`).

In aggiunta, la deep copy è stata utilizzata nella `put` per effettuare *copy-in* dell'argomento, nella `remove` per fare *copy-out* del file restituito (evitando che l'utente possa continuare ad editarlo anche dopo aver rinunciato all'accesso) e nella `copy` (vedi sotto).

Evitare l'uso della deep copy non è stato ritenuto possibile: ammettere solo tipi di dato immutabili avrebbe funzionato ma è stato ritenuto troppo limitante e non rispondente ai principi della *programmazione difensiva* (non sarebbe stato possibile imporre un controllo statico sulla mutabilità di `E`); prevedere un metodo apposito per la modifica del contenuto, con controllo dei permessi incorporato, sarebbe stato inefficace a meno di evitare la restituzione di qualunque puntatore al contenuto (cosa che avrebbe richiesto di rinunciare al metodo `get` e all'`iteratore`, ossia violare la consegna).

Precisazioni sulla sematica di alcuni metodi

La **copy** crea un nuovo file con contenuto identico ma separato (non una semplice duplicazione del puntatore: ne viene eseguita la *deep copy*), di cui il proprietario è l'utente invocatore e l'insieme dei collaboratori è inizializzato a vuoto. Si è scelta questa semantica perché l'alternativa (eseguire una *shallow copy* del contenuto, assegnando all'invocatore gli stessi diritti di cui godeva sul file originale anziché renderlo creatore) avrebbe avuto scarsa utilità pratica.

Si noti che la copy può essere eseguita anche da chi ha accesso al file in sola lettura: che lo stesso utente riceva l'accesso in scrittura del nuovo file è stato ritenuto coerente, poiché il contenuto del nuovo file non ha nessun legame con quello dell'originale (pur essendo identico): l'utente può modificare la sua copia, ma continua a non poter modificare l'originale.

Le due operazioni di **share** possono essere eseguite con successo solo dall'utente creatore del file, come indicato dalla consegna («utenti possono essere autorizzati dal proprietario ad accedere ai suoi file [...] altri non possono accedervi»).

La **remove** rimuove il file per sé stessi e non per tutti: ossia è stata intesa come "rinuncia" dell'utente al diritto d'accesso al file (a qualunque titolo lo avesse, compreso il caso in cui fosse il creatore); il file continua a esistere per tutti gli altri collaboratori. L'eliminazione effettiva dei dati viene eseguita solo quando l'ultimo collaboratore invoca la remove. Poiché non coinvolge gli altri utenti, il metodo può essere invocato anche da chi dispone solamente del permesso di lettura. Si è scelta questa semantica perché rendeva più semplice l'implementazione del metodo.

In **getSize** e **getIterator** si è inteso che i file su cui operare siano quelli a cui l'utente ha accesso, a qualunque titolo (non solo quelli di cui è creatore ma anche quelli condivisi con lui da altri utenti). I file che l'utente ha creato ma successivamente rimosso non vengono considerati, neanche se ancora accessibili da altri utenti con cui li aveva condivisi.

3 Descrizione delle due implementazioni

In fase di implementazione, nella scelta della struttura dati di supporto dove memorizzare i riferimenti ai file, si è dovuto tenere conto di alcune limitazioni derivanti dalla specifica: la presenza di file duplicati ha richiesto di scartare le collezioni che vietano gli elementi ripetuti; inoltre il fatto che l'identificatore dei file (cioè il contenuto) non sia unico, e che debba rimanere modificabile dopo l'inserimento, ha richiesto di scartare dizionari e mappe, che richiedono chiavi univoche e di valore costante. Rimanevano pertanto solo le strutture lineari, che nel *Java Collection Framework* sono solamente tre (`ArrayList`, `Vector` e `LinkedList`), tutte molto simili tra loro.

Per differenziare sufficientemente le due implementazioni, solo nella prima si è scelto di impiegare una struttura lineare (`ArrayList`), e nella seconda si è fatto ricorso a una mappa (`HashMap`) in cui le chiavi di ciascun file sono identificatori univoci generati appositamente. Però va detto che, anche nel secondo caso, non si hanno i benefici di efficienza tipici delle tabelle hash, perché la ricerca di un elemento al suo interno avviene comunque in modo lineare, effettuando una scansione completa della collezione (utilizzando l'iteratore); purtroppo fare diversamente non era possibile, dal momento che i metodi ricevono come argomento il contenuto, e la HashMap permette l'accesso diretto agli elementi solo indicando la chiave (non esiste un metodo di ricerca per valore).

Inoltre, per introdurre un ulteriore elemento di differenziazione tra le due implementazioni, è stato invertito il modo di associare le due componenti fondamentali dell'AdT, che nella prima è *utenti* $\rightarrow$ *file cui hanno accesso* e nella seconda *file* $\rightarrow$ *utenti autorizzati ad accedervi*.

Schema essenziale della prima implementazione: `ArraySecureFileContainer`

Una `HashMap` associa ogni username ad un oggetto "descrittore utente". Quest'ultimo memorizza la password e tre `ArrayList` che contengono rispettivamente i file creati, ricevuti in sola lettura e ricevuti in scrittura, sotto forma di riferimenti ad oggetti di tipo `E` (il contenuto).

Schema essenziale della seconda implementazione: `HashSecureFileContainer`

Utilizza due `HashMap`: la prima associa ogni username alla relativa password, la seconda memorizza valori di tipo "descrittore file" (la cui chiave è un numero progressivo intero). I descrittori memorizzano il contenuto del file (oggetto di tipo `E`), l'username dell'utente creatore, e due `HashSet` di stringhe, che rappresentano l'insieme degli utenti che hanno ricevuto accesso rispettivamente in sola lettura o in scrittura.

Si noti che l'`HashMap` che contiene i descrittori dei file, pur essendo concettualmente un insieme, non poteva essere realizzata con un `Set` perché esso non consente duplicati (come potrebbero esserlo due descrittori di file uguali per contenuto, utente creatore e utenti collaboratori) né consente modifiche al valore degli elementi (mentre noi desideriamo che il descrittore possa essere modificato in ogni momento). Un piccolo miglioramento di efficienza (non applicato perché non ritenuto fondamentale) si sarebbe potuto ottenere aggiungendo, nel campo "valore" della prima `HashMap`, anche un elenco dei file accessibili dall'utente (inteso come insieme delle corrispondenti chiavi nella seconda `HashMap`), in modo da restringere la ricerca del file ad un insieme più piccolo.