PROGETTO METODOLOGIE DI PROGRAMMAZIONE

NOME E COGNOME: JONATHAN COLOMBO
MATRICOLA: 7011579
PROGETTO: GESTIONE LIBRERIA

1. DESCRIZIONE DEL PROBLEMA

Si realizzi un sistema di gestione per una libreria online.
Una libreria desidera gestire elettronicamente il proprio catalogo delle tipologie di libri tramite opportune operazioni di amministrazione: inserimento di un libro, cancellazione di un libro, verifica se presente un libro, ricerca di un libro tramite titolo ecc...
Inoltre bisogna gestire il calcolo dei pagamenti delle varie tipologie di libro (audioBook, romanzi ecc...).
Gli utenti interessati ad un determinato libro devono essere notificati se il libro è disponibile o meno per poter effettuare un eventuale acquisto o noleggio.

2. SOLUZIONI CON PATTERN E DESCRIZIONE DIAGRAMMI UML

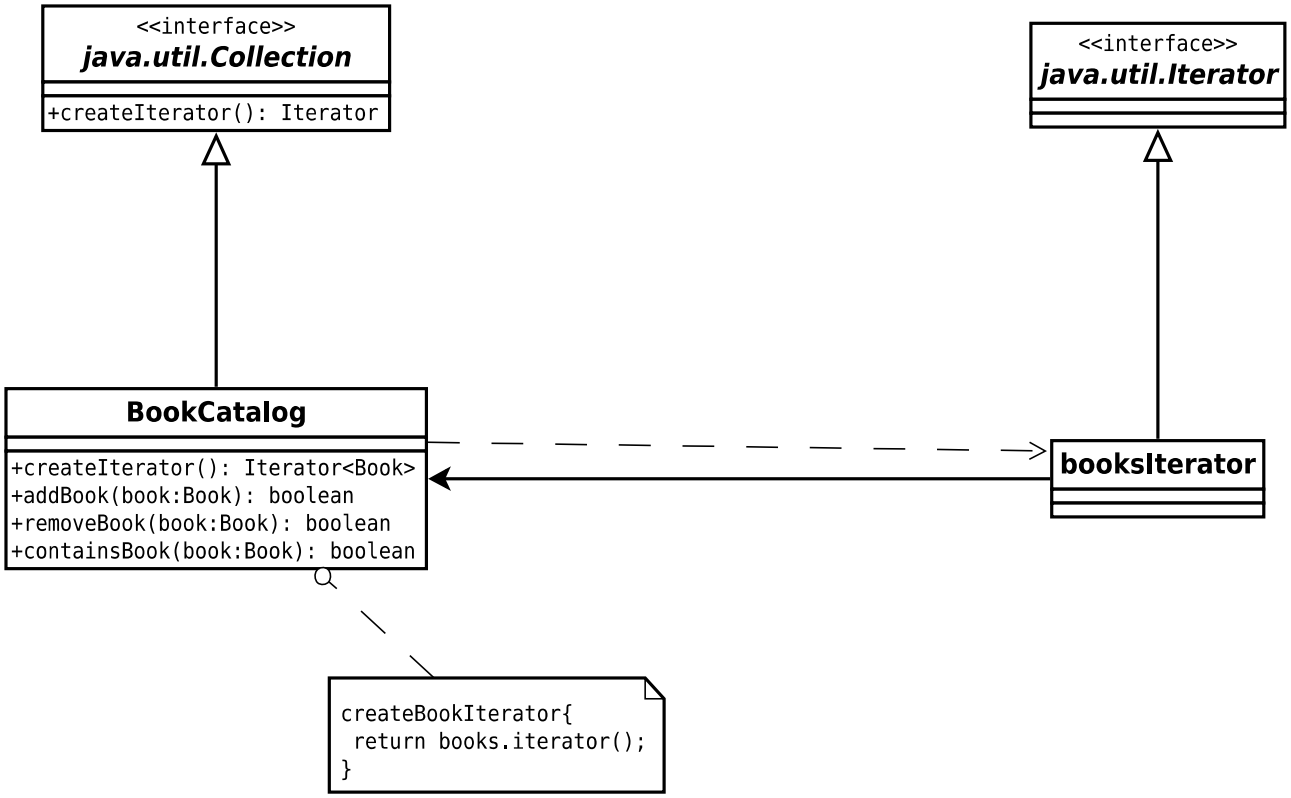
1. PROBLEMA

Si consideri il problema delle funzionalità di gestione dell'archivio dei libri.
Il client deve poter inserire, cancellare, visualizzare o verificare se è presente un qualsiasi libro.
Il pattern Iterator consente di attraversare e accedere agli elementi senza che venga esposta la struttura di riferimento.
Si possono effettuare anche operazioni interne ma in questo caso sono stati utilizzati i metodi dell'interfaccia Collection per aggiungere, rimuovere e verificare se è presente un libro nell'archivio.
Negli appositi test si verifica l'ordine in cui vengono inseriti i libri tramite un test con l'Iterator.

Alcuni metodi sono stati gestiti gestiti con opportune eccezioni per evitare di passare null agli argomenti.

PATTERN ITERATOR

Diagramma UML



Partecipanti

Iterator: interfaccia java.util.Iterator
- Specifica l’interfaccia per accedere e percorrere la collezione.

ConcreteIterator: oggetto che implementa l’interfaccia Iterator

Aggregate: interfaccia Collection
- Specifica una interfaccia per la creazione di oggetti Iterator

ConcreteAggregate: classe BookCatalog
- Crea e restituisce una istanza di Iterator

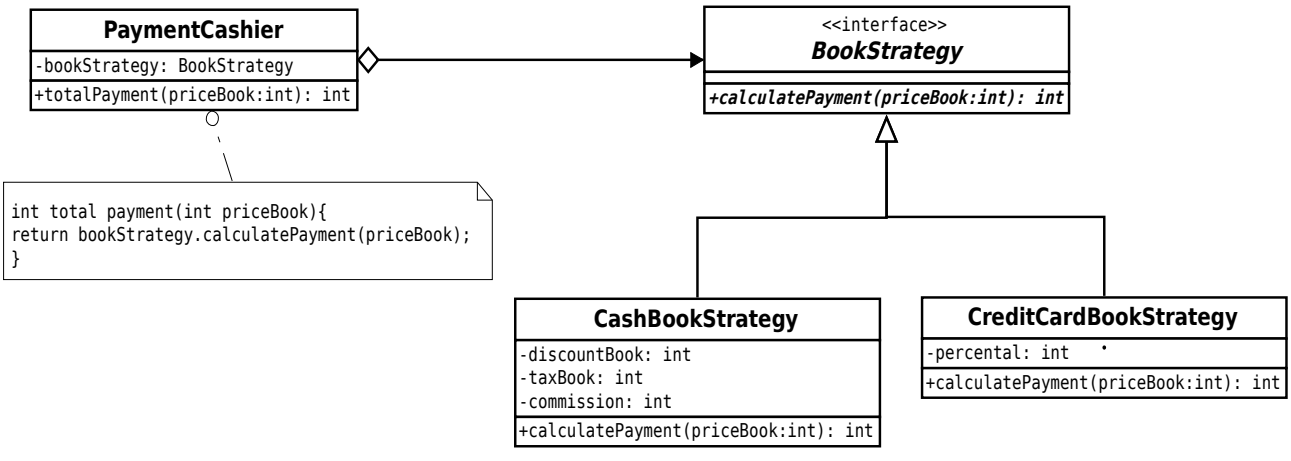
Implementazione

Java fornisce l’implementazione degli Iterator per le collezioni che implementano diretta o indirettamente l’interfaccia java.util.Iterator.
Il codice svolto per l’implementazione di questo pattern utilizza una Collection come collezione, e un Iterator per percorrerla.

2. PROBLEMA

Si consideri il problema della gestione dei pagamenti.
Ai libri che vengono pagati in contanti, il prezzo viene calcolato tramite tasse aggiuntive, commissioni e un opportuno sconto.
Per il calcolo dei libri pagati con carta di credito, lo sconto viene calcolato con una percentuale precisa senza tasse e commissioni aggiuntive.
Il pattern Strategy permette di definire una famiglia d’algoritmi.
Questo permette di modificare gli algoritmi in modo indipendente dai client che fanno uso di essi.
In questo caso le famiglie di algoritmi da definire sono due: calcola pagamento con contanti e con carta di credito.

PATTERN STRATEGY



Partecipanti

Strategy: interfaccia BookStrategy
- Dichiara una interfaccia comune per tutti gli algoritmi supportati.

Il Context utilizza questa interfaccia per invocare gli algoritmi definiti da ogni ConcreteStrategy.
ConcreteStrategy: classi CashBookStrategy e CreditCardBookStrategy
- Implementano gli algoritmi che usano l'interfaccia Strategy

Context: classe PaymentCashier
- Viene configurato con un oggetto ConcreteStrategy e mantiene un riferimento verso esso.
Specifica una interfaccia che consente alle Strategy di accedere ai propri dati.

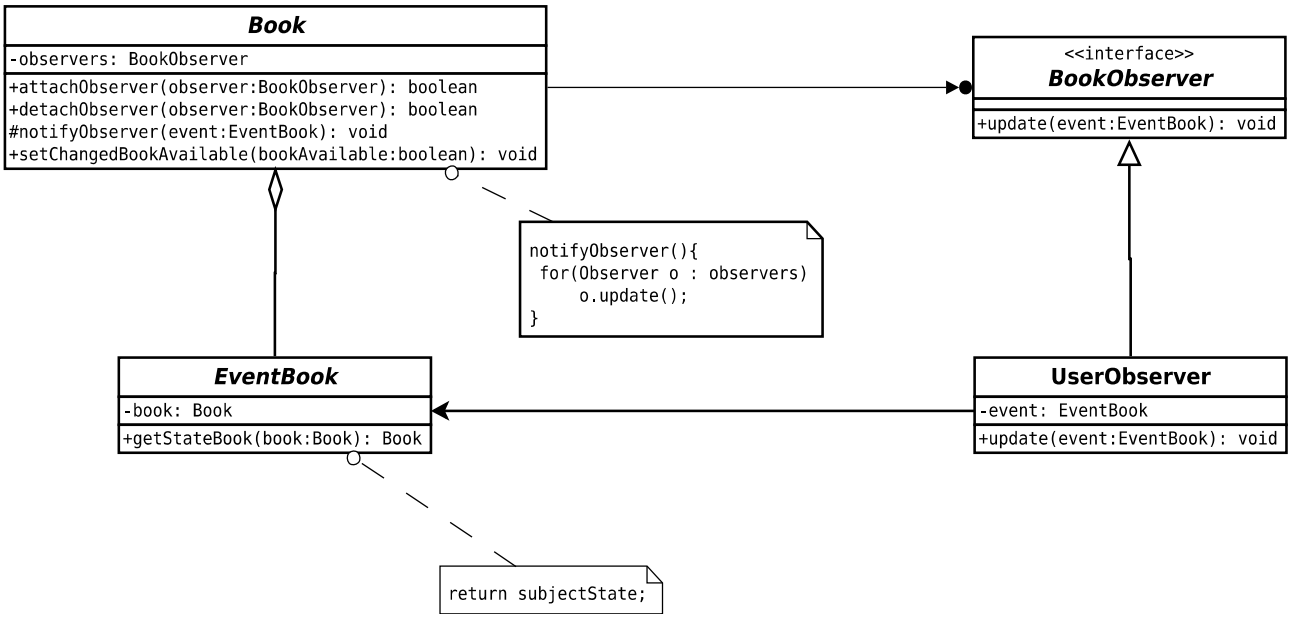
Implementazione

Si implementa la classe PaymentCashier che mantiene al suo interno un BookStrategy gestito con opportuno costruttore.
La modalità di pagamento è carico degli oggetti che implementano l'interfaccia BookStrategy.
L'oggetto ConcreteStrategy effettua la procedura di calcolo e prevede a impostarla con i vari parametri (commission, taxBook ecc…) ed essa viene invocata tramite il metodo totalPayment.

3. PROBLEMA

Gli utenti interessati ad un determinato libro devono essere notificati se il libro è presente o meno nella libreria.
Il pattern Observer consente la definizione di associazioni di indipendenza di molti oggetti verso di uno, in modo se quest'ultimo cambia il suo stato, tutti gli altri sono notificati e aggiornati automaticamente.
Il pattern assegna all'oggetto monitorato il ruolo di registrare al suo interno un riferimento ad altri oggetti che devono essere avvisati, e notificarli tramite l'invocazione a un loro metodo presente nell'interfaccia che devono implementare.
Questo modello può servire anche per comunicare eventi, in situazioni nelle quali non sia di interesse gestire una copia dello stato dell'oggetto osservato.

PATTERN OBSERVER
Diagramma UML



Partecipanti

Subject: classe Book
- Ha conoscenza dei propri Observer i quali possono esserci in numero illimitato.
Fornisce operazioni per aggiungere e cancellare oggetti Observer.
Fornisce operazioni per la notifica agli Observer.
Specifica una interfaccia per la notifica di eventi agli oggetti interessati in un Subject.

Observer: interfaccia BookObserver
- Specifica una interfaccia per la notifica di eventi agli oggetti interessati in un Subject.

ConcreteSubject: classe EventBook
- Invoca le operazioni di notifica ereditate dal Subject, quando devono essere informati i ConcreteObserver

ConcreteObserver: classe UserObserver
- Implementa l’operazione di aggiornamento dell’Observer

Implementazione

L’interfaccia BookObserver specifica le operazioni che devono implementare i ConcreteObserver.

Specifica il singolo metodo update che è il metodo che viene chiamato ogni volta che il Subject notifica un evento o un cambiamento nel proprio stato ai ConcreteObserver.
Il Subject passa come un parametro un riferimento a se stesso e un oggetto evento utile a trasferire informazioni aggiuntive.
Il Subject implementa questi metodi d'interesse:
- addObserver: registra l'Observer nel suo elenco interno di oggetti da notificare.
- removeObserver: elimina un Observer nel suo elenco interno di oggetti da notificare.
- setAvailable: imposta il valore per cambiare la disponibilità dell'oggetto.
- setChangedBookAvailable: se l'oggetto è stato segnato come cambiato, impostato dal metodo setAvailable, fa una notifica a tutti gli Observer per segnare che l'oggetto è cambiato tramite il metodo notifyObserver.
- notifyObserver(): se l'oggetto è stato segnato come cambiato, come indicato dal metodo setChangedBookAvailable, notifica tutti gli osservatori.
Si noti che il metodo notifyObserver viene invocata indipendentemente se si è registrato un cambio di stato nell'oggetto.
La classe UserObserver implementa l'interfaccia BookObserver, che ad ogni notifica di un cambiamento nel Subject, invoca un metodo su quest'ultimo per conoscere se il libro è disponibile o meno.

3) NOTE AGGIUNTIVE

Nelle varie classi di entità sono stati implementati metodi come hashCode(), equals() e toString a scopo didattico.
Opportuni metodi vengono gestiti tramite eccezioni per evitare di passare valori imprecisi o nulli.
Il progetto comprende la libreria aggiuntiva AssertJ per la scrittura dei JUnit Test.