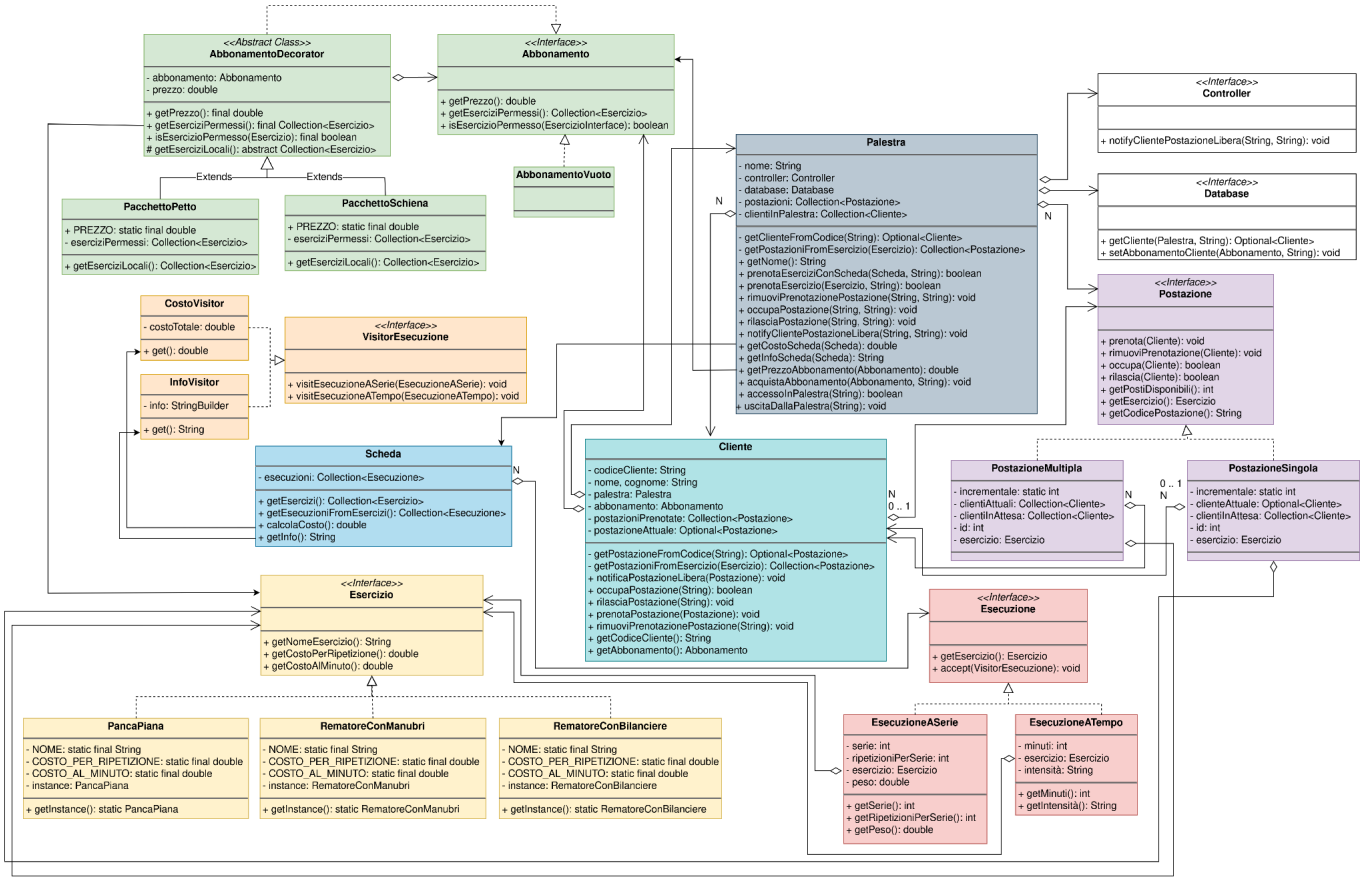
Relazione
Progetto di Metodologie di Programmazione

Lorenzo Bartolini
a.a 2022/2023

Abstract

Il progetto è incentrato sulla gestione di una palestra “smart”. Questo termine indica la natura fortemente tecnologica della palestra in questione, infatti, essa fornirà in particolare la possibilità di prenotare le postazioni che si intende utilizzare. La prenotazione non avviene nel futuro bensì nel presente. La persona che entra in palestra potrà presentare una scheda come piano di allenamento e il software procederà a trovare le postazioni necessarie a svolgere gli esercizi sulla scheda. Se la postazione è occupata si viene posti in attesa e notificati nel momento in cui si libera. Una volta ricevuta la notifica, il Cliente potrà inviare una richiesta di occupare la suddetta postazione, se non è stata già occupata da qualcun altro più rapidamente. Una volta che il Cliente riesce ad occupare una postazione, viene rimosso dall’attesa rispetto alle altre postazioni prenotate per lo stesso esercizio.
Si noti come da questa descrizione possa sorgere il problema della concorrenza. Questo però si ipotizza che venga gestito dal *Controller* (o da una terza parte) per disaccoppiare maggiormente il ruolo di *Model* dalla responsabilità di gestire la concorrenza.
La palestra è strutturata su Abbonamenti per l’ingresso e sul pagamento in base agli esercizi svolti. L’abbonamento è suddiviso in base agli esercizi che è possibile eseguire, infatti, prima di entrare in palestra, è necessario convalidare la Scheda ovvero verificare che gli esercizi che si intende svolgere siano ammessi con il proprio abbonamento. Se la Scheda fosse corretta allora si procede al pagamento. Una Scheda è formata da esecuzioni di esercizi, ovvero dall’esercizio e dalle istruzioni di come svolgerlo. Il Cliente può verificare il costo della Scheda oppure riceverne le informazioni dettagliate.
Sono state previste due possibili implementazioni concrete dell’interfaccia associata all’Esecuzione, per Ripetizioni e a Tempo.
Le postazioni prenotabili si dividono in postazioni Singole e Multiple, come suggerisce il nome le postazioni multiple permettono a più persone di lavorare contemporaneamente.

UML Completo



Architettura: MVC

Il sistema realizza un'architettura MVC, in particolare è stato sviluppato il *Model*. La componente di *View* non viene presa in considerazione nel codice in quanto non è direttamente collegata al *Model*.

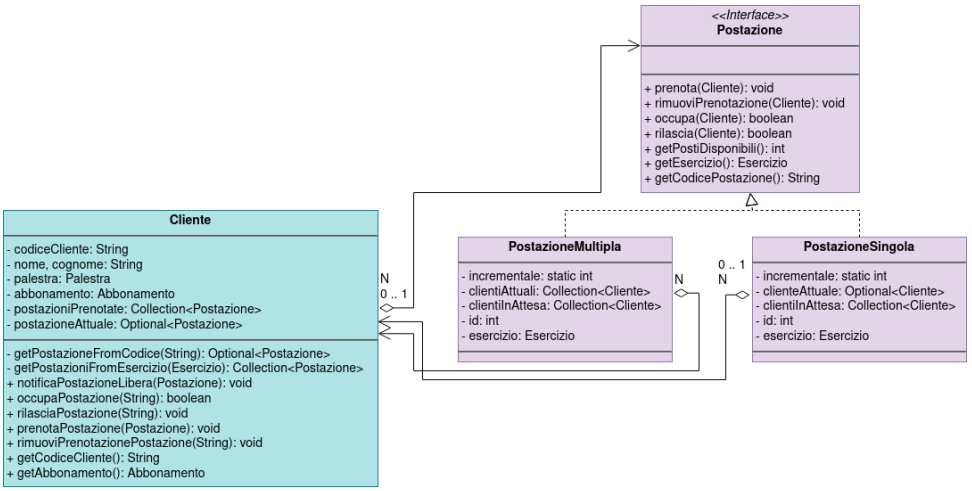
Quest'ultimo infatti deve dialogare con il *Controller*, sistema preposto alla gestione e serializzazione delle richieste dirette e provenienti dalla *View*, e con la base di dati grazie all'interfaccia *Database* che viene interrogata dal *Model,* in particolare dalla classe Palestra.

Sia *Database* che *Controller* non sono stati implementati concretamente perché non di particolare interesse ai fini del progetto.

Pattern usati e scelte progettuali

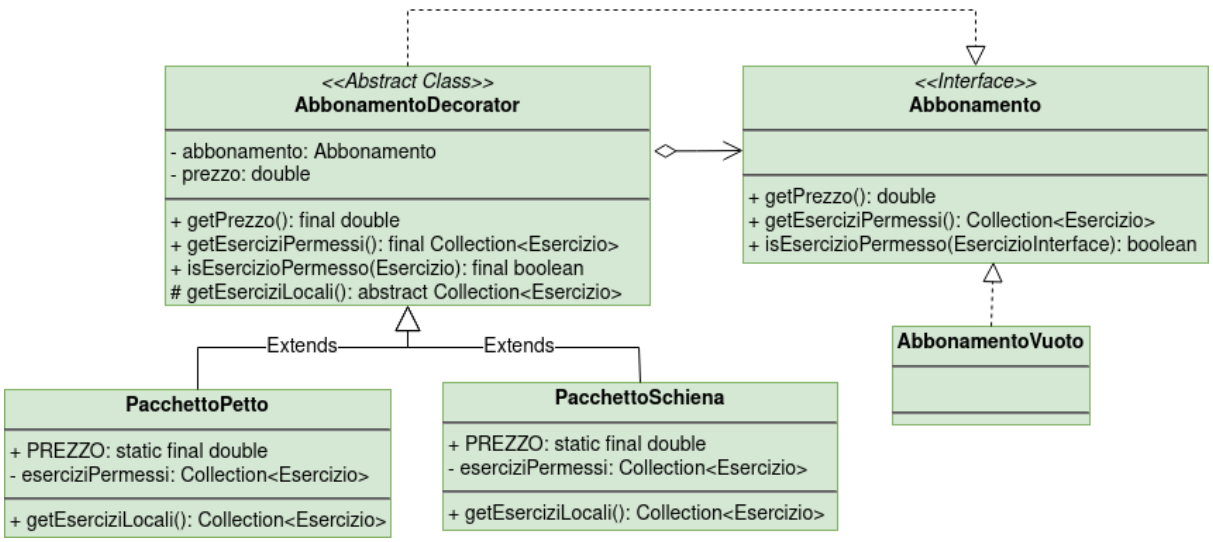
Di seguito saranno approfonditi i pattern e le relative scelte progettuali che ne hanno motivato lo sviluppo.

Observer:



Il pattern Observer è il pattern centrale del progetto. Esprime il modo in cui il Cliente effettua la prenotazione delle Postazioni.
Le Postazioni hanno il ruolo di Subject, ovvero di oggetti osservati e di cui è di interesse lo stato.
I Clienti invece svolgono il ruolo di Observer, ovvero di oggetti che osservano i Subject e rimangono in attesa di una notifica che lo stato del Subject è cambiato. Si noti che non è stata realizzata un'interfaccia per gli Observer in quanto il progetto non prevede altri Observer se non i Clienti.
Nel progetto lo stato di interesse è il fatto che la postazione sia libera per essere occupata.
La notifica è svolta automaticamente all'interno delle Postazioni concrete in due casi: il primo in cui il cliente prenota una postazione libera e il secondo in cui un cliente rilascia la postazione che ha occupato fino a quel momento notificando i restanti Clienti in attesa.
Quando il Cliente viene notificato che una Postazione è libera, inoltra il messaggio a Palestra che lo inoltrerà a *Controller*. A quel punto la persona potrà effettuare un tentativo di occupare la postazione che si è appena liberata. Nel caso in cui ci sia almeno un posto disponibile, ci riuscirà altrimenti rimarrà in attesa.
Si noti che il metodo di Cliente che occupa una postazione esegue un'altra operazione; se l'occupazione va a buon fine rimuove dalle postazioniPrenotate tutte le postazioni relative al medesimo esercizio.
Per scelta progettuale è stato previsto infatti che se un Cliente intende prenotare un esercizio, verranno prenotate tutte le postazioni che offrono quell'esercizio per minimizzare il tempo di attesa. Quando finalmente riuscirà ad occuparne una, le altre saranno rimosse dall'osservazione.
Si noti che la prenotazione delle postazioni non dipende dalle esecuzioni degli esercizi presenti nella scheda ma dagli esercizi presenti in essa. Sarà cura della persona effettuare tutte le esecuzioni relative allo stesso esercizio sulla stessa postazione oppure prenotare manualmente la postazione per una seconda volta.

Decorator, Template Method e Chain of Responsabilities:



Questi tre pattern saranno discussi insieme in quanto parte della stessa porzione di progetto riguardante la modellazione degli Abbonamenti.

E' stato deciso di progettare gli Abbonamenti con una struttura modulare in modo tale da poter aggiungere funzionalità dinamicamente. In questo modo è stato possibile evitare di avere una classe separata per ogni combinazione di pacchetti.

Le operazioni che interessano il Decorator sono quelle di calcolare il prezzo, ottenere tutti gli esercizi permessi e verificare che un esercizio sia permesso. Queste operazioni sono state definite una volta per tutte nella classe astratta AbbonamentoDecorator.

Per garantire un buon grado di versatilità è stato usato il pattern Template Method.

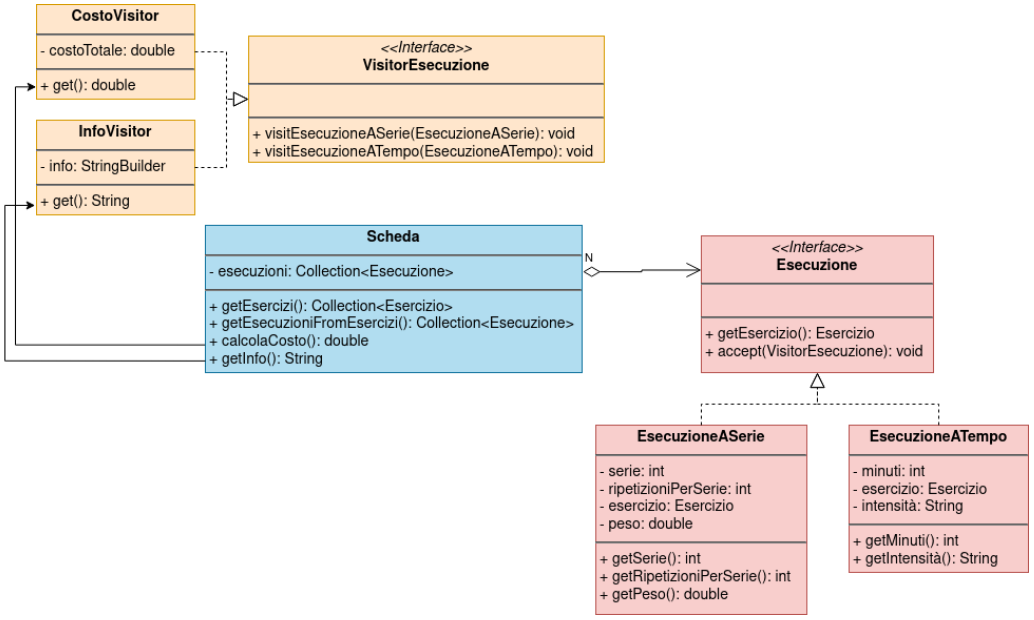
Grazie a questo pattern è stato permesso di delegare alle classi che estendono il Decorator l'operazione di ottenere gli esercizi che i singoli pacchetti offrono, tramite il metodo getEserciziLocali(). Questo metodo viene usato all'interno dei due Template Methods: getEserciziPermessi(), in modo tale da concatenare gli esercizi del pacchetto con quelli dell'oggetto decorato, e isEsercizioPermesso(), che controlla se l'esercizio da controllare sia permesso.

Il pattern Chain of Responsabilities è stato impiegato in isEsercizioPermesso() per rispondere ad un possibile problema legato alle performance della ricerca su una collezione. L'approccio iniziale potrebbe essere quello di implementare tale metodo prendendo prima tutti gli esercizi permessi e successivamente verificare che l'esercizio da controllare ne faccia parte, tramite il metodo contains().

Con collezioni particolarmente grandi questo potrebbe essere un problema, è stato deciso quindi di delegare ai singoli pacchetti la ricerca sulla personale collezione di esercizi permessi. Il primo pacchetto che effettivamente contiene l'esercizio da testare interrompe la catena risparmiando tempo e risorse.

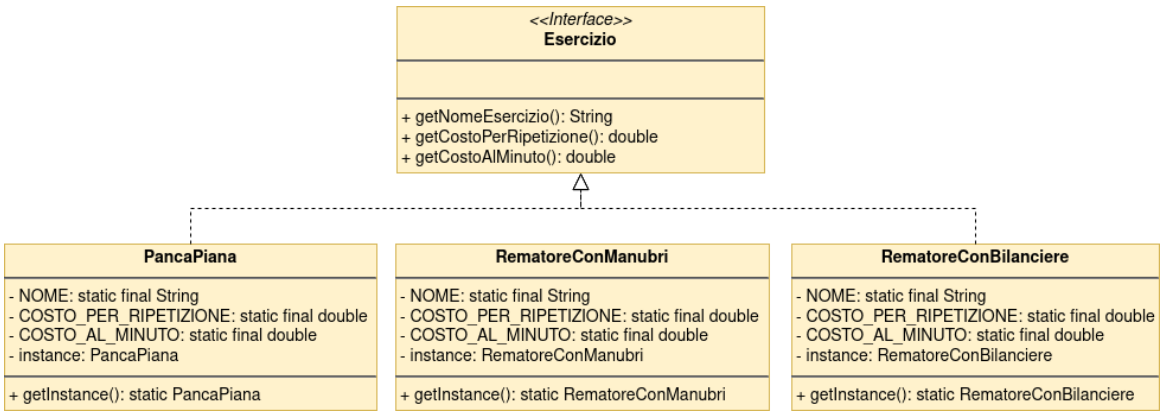
Nonostante la perdita di flessibilità legata all'aver codificato in maniera definitiva questi due algoritmi abbiamo guadagnato nella facilità di implementare nuovi pacchetti che si integrano perfettamente con gli abbonamenti già presenti. Intendo far notare come è prevista una classe AbbonamentoVuoto che funge da base per le decorazioni. Inoltre assicura che nel Chain of Responsabilities, la richiesta venga sempre gestita poichè ritorna false.

Visitor:



E' stato deciso di introdurre Visitor per facilitare le operazioni riguardanti le esecuzioni all'interno della scheda.
Sono presenti due tipi di esecuzione molto diverse tra loro sulle quali è necessario eseguire le medesime operazioni (calcolo del costo e ottenimento delle informazioni) ma in modalità diverse.
Voglio far notare come si sarebbe potuto implementare tali operazioni all'interno dell'interfaccia Esecuzione, ma è stato preferito evitare ciò per mantenere separato il concetto di Esecuzione da quello legato alle sue operazioni. Se avessimo implementato le varie operazioni direttamente nelle classi figlie avremmo definito una volta per tutte l'algoritmo. Se in futuro volessimo calcolare differentemente il costo della Scheda saremmo costretti a modificare le esecuzioni, questo ha poco senso e porterebbe solo a problemi.
E' stato deciso di utilizzare un Visitor Void per lasciare più libertà possibile allo sviluppatore.
Si noti che i visitor concreti vengono istanziati direttamente nei metodi della classe Scheda.
In futuro se si volesse cambiare o aggiungere operazioni basterà creare un nuovo Visitor Concreto.

Singleton



Per quanto riguarda la struttura degli Esercizi l'interfaccia è molto semplice e comprende solo i metodi necessari al costo. Infatti i singoli esercizi figli ricoprono nel progetto il solo ruolo astratto di esercizio. Non avendo alcuno stato ed essendo usati in ogni punto del progetto è stato preferito implementare il pattern Singleton per quanto riguarda la creazione degli oggetti. Questo pattern permette di evitare oggetti duplicati che occuperebbero memoria inutilmente.

Test e note finali

Per prima cosa sono state realizzate due classi Mock per Controller e per Abbonamento.
Il ControllerMock si occupa di memorizzare il numero di notifiche che riceve, questo è stato necessario per testare il sistema di invio di notifiche verso i Clienti.
L'AbbonamentoMock è stato realizzato per restituire sempre il valore passato al costruttore come risultato del metodo isEsercizioPermesso(), questo si è rivelato utile per testare i metodi della Palestra per prenotare gli esercizi di una scheda, in questo modo si è testato sia il cammino corretto in cui tutti gli esercizi erano permessi sia quello non corretto in cui gli esercizi non erano permessi.
Tramite lo strumento Coverage As di Eclipse è possibile vedere la percentuale di codice testato, osservando che tutti i metodi interessanti vengono testati. Non sono stati testati, per ovvie ragioni, i getter e alcuni metodi che non facevano altro che inoltrare ad un altro oggetto l'esecuzione.
Alcuni attributi e metodi sono stati resi Package-Private per facilitare i test.
Saranno ora analizzati in dettaglio i test svolti:

- Test per Abbonamento
    - Funzionalità isEsercizioPermesso, testata prima con un solo pacchetto che decora l'abbonamento vuoto e poi aggiungendo un secondo pacchetto a quello precedente e verificando che gli esercizi che erano presenti prima lo fossero ancora insieme a quelli del nuovo pacchetto appena aggiunto.
    - Funzionalità getEserciziPermessi, testata allo stesso modo verificando che la collezione risultante contenesse esattamente gli oggetti aspettati.
    - Funzionalità getPrezzo, testata allo stesso modo delle precedenti.
    - Infine è stato testato che non sia possibile decorare un oggetto null.
- Test per Postazioni
    - Funzionalità di Occupazione Postazione Singola, testata creando due postazioni identiche e prenotandole entrambe attraverso un Cliente. Si procede tentando di occuparne una e verificando che risulti occupata dal cliente con cui è stata effettuata la prenotazione. Inoltre si è verificato che i clientiInAttesa di entrambe le postazioni fossero vuoti in quanto, nel momento dell'occupazione di una postazione, vengono rimosse tutte le prenotazioni di altre postazioni che offrono lo stesso esercizio.
    - Funzionalità di Occupazione postazione Singola cammino non corretto, testata predisponendo un'unica postazione e tentando di occuparla con due Clienti diversi.
    - Funzionalità di Rilascio postazione Singola, testata mettendo in attesa tre clienti. Uno di essi occupa la postazione e successivamente la rilascia. Mediante l'utilizzo della classe ControllerMock viene testato che vengano inoltrate tutte le notifiche. Si noti che una notifica viene lanciata anche nel momento in cui si prenota una postazione libera e non solo durante il rilascio.
    - Le medesime funzionalità sono state testate anche nel caso della postazione multipla con le stesse modalità.
- Test per Palestra
    - Test di getClienteFromCodice, testato inserendo manualmente un Cliente tra i clientiInPalestra e ricercandolo tramite il suo codice. Viene effettuato un test

che ci aspettiamo vada a buon fine e uno che fallisca. In questo modo siamo sicuri che il metodo sia effettivamente corretto in base all'input.
    - Test di getPostazioniFromEsercizio, testato inserendo nella palestra alcune postazioni di PancaPiana e altre di RematoreConManubri. Eseguendo il metodo con diversi input restituisca correttamente le postazioni, anche nel caso di postazioni non presenti nella palestra.
    - Test di prenotazione esercizi tramite Scheda, testato inserendo nella palestra una serie di postazioni divise in due gruppi: un gruppo che ci aspettiamo venga prenotato e un'altro no. Il cliente è inizializzato con AbbonamentoMock che permette ogni tipo di esercizio. La scheda utilizzata contiene esercizi di PancaPiana e RematoreConManubri e non RematoreConBilanciere. Viene quindi chiamato il metodo prenotaEserciziConScheda e ci assicuriamo che vada a buon fine e che tutte le notifiche, derivanti dalla prenotazione di postazioni libere, vengano inoltrate con successo al Controller. Infine verifichiamo che le postazioni siano tra quelle effettivamente prenotate.
    - Test di prenotazione tramite Scheda cammino non corretto, testato allo stesso modo con la differenza che l'abbonamento non permette di eseguire alcun esercizio. Testiamo che il tentativo di prenotazione non vada a buon fine.
    - I medesimi test sono stati effettuati anche nel caso di prenotazione di un esercizio senza la Scheda.
- Test per Scheda
    - Vengono testati i metodi privati getEsercizi e getEsecuzioniFromEsercizio con modalità simili ai metodi privati di Palestra.
    - Test di Calcolo Costo, testato creando una scheda e calcolandone il costo verificando che fosse corretto. Si ricordi che, benchè possa sembrare un'operazione banale, questo metodo di Scheda coinvolge il Visitor quindi è un test fondamentale per il corretto funzionamento di questa parte del progetto.
    - Test di Info Scheda, testato similmente al precedente.