



UNIVERSITÀ  
DEGLI STUDI  
FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE  
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

---

## **Progetto Piattaforma per la Gestione di Videogiochi**

---

*Autore:*  
Samuele Lattanzi

*N° Matricola:*  
7048716

*Corso principale:*  
Ingegneria del Software

*Docente corso:*  
Enrico Vicario

# Indice

<b>1</b>	<b>Introduzione</b>	<b>3</b>
1.1	Statement del progetto . . . . .	3
1.2	Architettura e strumenti . . . . .	3
1.3	Relazioni tra i pacchetti . . . . .	3
<b>2</b>	<b>Documentazione</b>	<b>5</b>
2.1	Use Case Diagram . . . . .	5
2.2	Use Case Template . . . . .	6
2.3	Mock-Ups . . . . .	8
2.4	Class Diagram . . . . .	11
2.5	ER Diagram . . . . .	11
<b>3</b>	<b>Implementazione</b>	<b>12</b>
3.1	Domain Model . . . . .	12
3.1.1	Utente . . . . .	12
3.1.2	Videogioco . . . . .	12
3.1.3	Carrello . . . . .	12
3.1.4	Libreria . . . . .	12
3.1.5	Abbonamento . . . . .	13
3.1.6	StatisticheVideogioco . . . . .	13
3.1.7	Achievement . . . . .	13
3.1.8	Admin . . . . .	13
3.2	Business Logic . . . . .	13
3.2.1	UtenteController . . . . .	13
3.2.2	CarrelloController . . . . .	14
3.2.3	LibreriaController . . . . .	16
3.2.4	CatalogoController . . . . .	17
3.2.5	StatisticheController . . . . .	18
3.2.6	AchievementController . . . . .	18
3.2.7	AdminController . . . . .	19
3.3	ORM (Object-Relational Mapping) . . . . .	19
3.3.1	DatabaseConnection . . . . .	19
3.3.2	UtenteDAO . . . . .	19
3.3.3	VideogiocoDAO . . . . .	20
3.3.4	CarrelloDAO . . . . .	21
3.3.5	LibreriaDAO . . . . .	21
3.3.6	AbbonamentoDAO . . . . .	22
3.3.7	StatisticheVideogiocoDAO . . . . .	23
3.3.8	AchievementDAO . . . . .	23
3.3.9	AdminDAO . . . . .	24
3.4	Database . . . . .	24
<b>4</b>	<b>Test</b>	<b>25</b>
4.1	Business Logic Test . . . . .	25
4.1.1	UtenteControllerTest . . . . .	25
4.1.2	CatalogoControllerTest . . . . .	26
4.1.3	CarrelloControllerTest . . . . .	27
4.1.4	LibreriaControllerTest . . . . .	29
4.1.5	StatisticheControllerTest . . . . .	31
4.1.6	AchievementControllerTest . . . . .	32
4.1.7	AdminControllerTest . . . . .	33
4.2	Domain Model Test . . . . .	34
4.2.1	CarrelloTest . . . . .	34
4.3	Test del Pacchetto ORM . . . . .	36
4.3.1	VideogiocoDAOTest . . . . .	36
4.3.2	CarrelloDAOTest . . . . .	37
4.3.3	LibreriaDAOTest . . . . .	39

## Elenco delle figure

1	Use Case Diagram - Utente . . . . .	5
2	Use Case Diagram - Admin . . . . .	5
3	Mockup raffigurante un prototipo del menu iniziale dell'applicazione . . . . .	8
4	Mockup raffigurante un prototipo del menu di login . . . . .	9
5	Mockup raffigurante un prototipo della schermata principale per l'utente . . . . .	9
6	Mockup raffigurante un prototipo della visualizzazione del catalogo di videogiochi . . . . .	9
7	Mockup raffigurante un prototipo della visualizzazione del carrello . . . . .	10
8	Mockup raffigurante un prototipo della visualizzazione della libreria . . . . .	10
9	Mockup raffigurante un prototipo del menu dell'admin . . . . .	10
10	Diagramma di Classe, con suddivisione nei pacchetti Domain Model, Business Logic, ORM . . .	11
11	Diagramma ER: Rappresentazione della struttura delle tabelle create dal DBMS . . . . .	11

# 1 Introduzione

## 1.1 Statement del progetto

L'obiettivo del progetto è lo sviluppo di un'applicazione per la gestione di un sistema di vendita di videogiochi. L'applicazione offre una piattaforma attraverso cui gli utenti possono esplorare un catalogo di videogiochi, effettuare ricerche basate su criteri come genere, piattaforma e data di uscita, e aggiungere titoli a un carrello virtuale per l'acquisto. Una volta acquistati, i giochi vengono inseriti in una libreria personale dell'utente, che consente di gestire i titoli posseduti, permettendo di installarli, disinstallarli o avviarli per giocare. Durante le sessioni di gioco, l'applicazione registra statistiche, come il tempo totale di gioco e permette di sbloccare achievement associati ai videogiochi. Il sistema prevede inoltre due tipologie di abbonamento mensile, Gold e Silver, che offrono agli utenti sconti esclusivi e accesso a una selezione di giochi gratuiti. Per aver accesso a questi abbonamenti e poter comprare i videogiochi che ha inserito nel carrello, un utente ha un fondo, che può ricaricare. È incluso un ruolo amministratore (Admin), con funzionalità dedicate alla gestione del catalogo, come l'aggiunta, la rimozione e la modifica dei giochi, nonché la selezione dei titoli gratuiti per gli abbonati e la possibilità di aggiungere, rimuovere e modificare gli achievement associati ai giochi.

## 1.2 Architettura e strumenti

L'applicazione è stata progettata seguendo un'architettura modulare, suddivisa in pacchetti distinti per garantire chiarezza, manutenibilità e scalabilità del codice. I principali componenti architetturali sono:

- **Domain Model:** definisce le entità principali del sistema, come utenti, videogiochi, carrello, libreria, statistiche e achievement, rappresentando la struttura dei dati.
- **Business Logic:** contiene la logica applicativa, gestendo operazioni come la ricerca dei videogiochi, l'elaborazione degli acquisti, la gestione delle statistiche di gioco e le funzionalità amministrative.
- **ORM (Object-Relational Mapping):** utilizzato per mappare gli oggetti del domain model alle tabelle del database, semplificando le operazioni di persistenza.
- **JDBC:** impiegato per il collegamento al database relazionale PostgreSQL, garantendo un'interazione efficiente e sicura con i dati.

Il database scelto per il progetto è **PostgreSQL**. L'interfaccia utente è stata implementata come una Command Line Interface (**CLI**), offrendo un'interazione semplice e diretta con il sistema.

Per lo sviluppo, la documentazione e il design, sono stati utilizzati i seguenti strumenti:

- **StarUML:** impiegato per la creazione dei diagrammi UML, inclusi i diagrammi degli *use case* e dei diagrammi di classe, utili per definire la struttura e il comportamento del sistema.
- **Figma:** utilizzato per la progettazione dei mockup dell'interfaccia utente, anche se l'implementazione finale si è concentrata sulla CLI.
- **draw.io:** impiegato per la creazione dei mockup.
- **Eclipse IDE:** ambiente di sviluppo integrato (IDE) scelto per la scrittura, la gestione e il debug del codice sorgente.
- **JUnit 5:** utilizzato per l'implementazione dei test.

## 1.3 Relazioni tra i pacchetti

Le relazioni tra i pacchetti del sistema sono organizzate secondo un'architettura a strati, dove ogni livello interagisce con quello sottostante per garantire modularità e separazione delle responsabilità. Di seguito è descritta la relazione tra i vari componenti:

- **Interfaccia (CLI):** L'interfaccia a riga di comando (CLI) rappresenta il punto di contatto con l'utente. Essa invoca i metodi esposti dalla Business Logic per gestire le interazioni tra utente e sistema.
- **Business Logic:** Utilizza le entità definite nel domain model e si affida all'ORM per le operazioni di persistenza, evitando interazioni dirette con il database.

- **ORM (Object-Relational Mapping):** L'ORM funge da intermediario tra la business logic e il database, mappando le entità del domain model alle tabelle del database relazionale. Questo strato traduce le operazioni sui dati (come creazione, lettura, aggiornamento ed eliminazione) in query SQL, senza che la business logic debba gestire direttamente i dettagli del database.
- **JDBC (Java Database Connectivity):** JDBC fornisce il collegamento tra l'ORM e il database PostgreSQL. Attraverso le API di JDBC, l'ORM invia query SQL al database e riceve i risultati, garantendo un'interazione efficiente e sicura con l'RDBMS.
- **RDBMS (PostgreSQL):** Il sistema di gestione del database relazionale (RDBMS) PostgreSQL memorizza i dati del sistema. È il livello più basso e risponde esclusivamente alle richieste effettuate tramite JDBC.

## 2 Documentazione

### 2.1 Use Case Diagram

In questa sezione vengono presentati use case diagram, che descrivono le interazioni tra gli attori del sistema (Utente, Admin) e le funzionalità principali dell'applicazione.

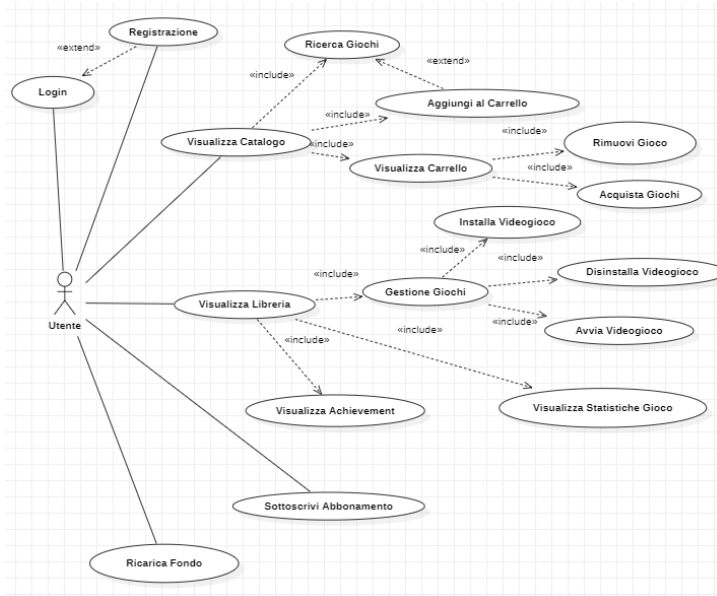


Figura 1: Use Case Diagram - Utente

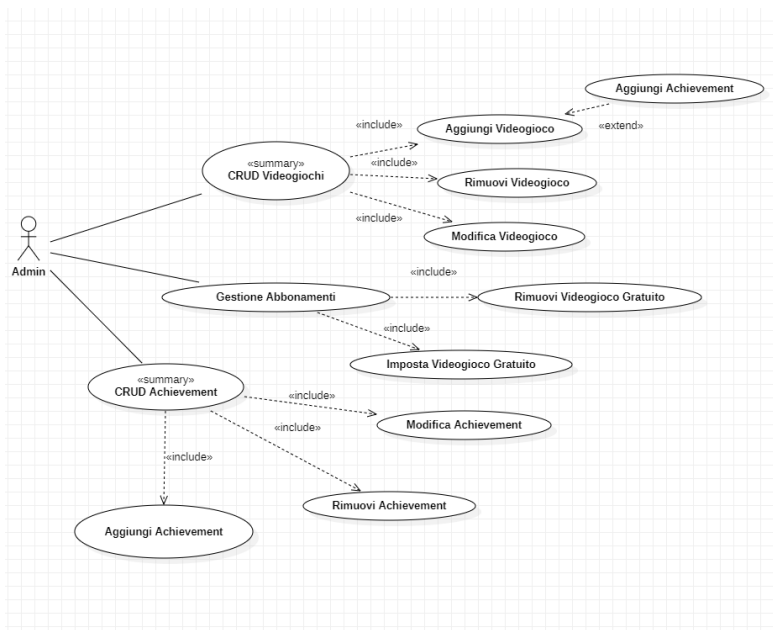


Figura 2: Use Case Diagram - Admin

Il diagramma degli use case, mostrato nella Figura 1, illustra le principali funzionalità che riguardano l'utente, mentre quello mostrato nella Figura 2 fa riferimento all'admin.

## 2.2 Use Case Template

In questa sezione vengono presentati i template relativi ad i casi d'uso principali del progetto.

Use Case 1: Acquistare un videogioco (UC-001)	
<b>Livello</b>	User Goal
<b>Descrizione</b>	Consentire a un utente autenticato di selezionare videogiochi dal catalogo, aggiungerli al carrello e completare l'acquisto, aggiornando la libreria e il fondo.
<b>Attori</b>	Utente
<b>Preconditions</b>	L'utente deve essersi autenticato nel sistema e si deve trovare nel menu principale
<b>Steps</b>	<ol style="list-style-type: none"> <li>1. L'utente seleziona l'opzione per accedere al catalogo (dal menu in Mock-up 5)</li> <li>2. L'utente sceglie tra l'opzione di visualizzazione o di ricerca</li> <li>3. L'utente visualizza l'elenco completo di tutti i giochi acquistabili, oppure solo quelli filtrati attraverso la ricerca (Mock-up 6, Test 29)</li> <li>4. L'utente seleziona il videogioco che vuole acquistare e lo aggiunge al carrello</li> <li>5. L'utente può visualizzare il carrello e costo totale (Mock-up 7)</li> <li>6. L'utente esegue l'acquisto dei giochi nel carrello.</li> </ol>
<b>Svolgimenti alternativi</b>	<ul style="list-style-type: none"> <li>• 4a. Se l'utente sceglie un gioco che ha già acquistato viene notificato con un messaggio e riprova (Test 30)</li> <li>• 6a. Se l'utente accede ad un carrello vuoto riceve un messaggio e torna al catalogo</li> <li>• 6b. Se l'utente ha un fondo insufficiente viene notificato e non può completare l'acquisto (Test 28)</li> </ul>
<b>Postconditions</b>	Il carrello viene svuotato, i giochi sono spostati nella libreria personale e salvati nel sistema e il fondo viene aggiornato.

Tabella 1: Template che descrive il caso d'uso per l'acquisto di videogiochi da parte di un utente

Use Case 2: Gestire la libreria personale (UC-002)	
<b>Livello</b>	User Goal
<b>Descrizione</b>	Consente a un utente autenticato di visualizzare, installare, disinstallare, avviare giochi e consultare statistiche e achievement.
<b>Attori</b>	Utente
<b>Preconditions</b>	L'utente ha effettuato il login e possiede giochi acquistati
<b>Steps</b>	<ol style="list-style-type: none"> <li>1. L'utente seleziona l'opzione per accedere alla propria libreria di giochi (Mock-up 5)</li> <li>2. Il sistema mostra i giochi acquistati dall'utente, con la loro descrizione e stato di installazione (Mock-up 8)</li> <li>3. L'utente seleziona di installare o disinstallare un gioco</li> <li>4. Il sistema aggiorna lo stato di installazione o il tempo di gioco associato al videogioco selezionato.</li> </ol>
<b>Svolgimenti alternativi</b>	<ul style="list-style-type: none"> <li>• 3a. Se la libreria è vuota l'utente riceve un messaggio per avvertirlo di non poter compiere azioni.</li> <li>• 3b. L'utente seleziona l'opzione di avviare un videogioco e giocarlo <ul style="list-style-type: none"> <li>– 3b.1 L'utente seleziona quanto tempo vuole giocare al videogioco.</li> <li>– 3b.2 L'utente simula di giocare a un videogioco e viene aggiornato il tempo di gioco.</li> </ul> </li> <li>• 3c. L'utente seleziona l'opzione di visualizzare statistiche e achievement relativi ad un videogioco (Test 32) <ul style="list-style-type: none"> <li>– 3c.1 Il sistema mostra le statistiche associate al gioco.</li> <li>– 3c.2 se non ci sono achievement associati ad un gioco il sistema manda un messaggio. (Test 33)</li> </ul> </li> <li>• 3d. Se l'utente seleziona di installare un gioco già installato o viceversa il sistema glielo impedisce e manda un messaggio (Test 31)</li> </ul>
<b>Postconditions</b>	Il sistema aggiorna lo stato di installazione o il tempo di gioco associato al videogioco selezionato e le modifiche sono salvate nel database

Tabella 2: Template che descrive il caso d'uso per la gestione della libreria e dei giochi contenuti in essa.

Use Case 3: Gestire il catalogo come amministratore (UC-003)	
<b>Livello</b>	Summary
<b>Descrizione</b>	Consente all'amministratore di aggiungere, modificare, eliminare videogiochi e achievement e gestire lo stato gratuito dei videogiochi del catalogo.
<b>Attori</b>	Admin
<b>Preconditions</b>	L'amministratore ha effettuato il login con credenziali valide
<b>Steps</b>	<ol style="list-style-type: none"> <li>1. L'admin ha accesso al menu da amministratore (Mock-up 9)</li> <li>2. L'admin seleziona azione: aggiungi, modifica, elimina videogioco o achievement oppure imposta/rimuovi gratuito (Test 29 e Test 33).</li> <li>3. Il sistema esegue l'operazione e mostra la conferma.</li> </ol>
<b>Svolgimenti alternativi</b>	<ul style="list-style-type: none"> <li>• 1a. Se le credenziali non sono valide viene mandato un messaggio di errore (Test 34)</li> <li>• 2a. Se i dati immessi dall'admin durante la creazione o la modifica non sono validi riceve messaggio di errore</li> </ul>
<b>Postconditions</b>	Il catalogo viene aggiornato nel database e le modifiche ai videogiochi e agli achievement salvate

Tabella 3: Template che descrive il caso d'uso per la gestione delle operazioni CRUD effettuate dall'admin

Use Case 4: Gestire gli abbonamenti (UC-004)	
<b>Livello</b>	User Goal
<b>Descrizione</b>	Consente a un utente autenticato di scegliere e attivare un abbonamento (Gold o Silver), che offre sconti sui giochi e accesso a giochi gratuiti.
<b>Attori</b>	Utente
<b>Preconditions</b>	L'utente ha effettuato il login con credenziali valide.
<b>Steps</b>	<ol style="list-style-type: none"> <li>1. L'utente seleziona l'opzione per gestire gli abbonamenti dal menu utente (Mock-up 5)</li> <li>2. Il sistema mostra lo stato attuale dell'abbonamento (nessuno, Gold o Silver) e chiede che tipo di abbonamento attivare.</li> <li>3. L'utente seleziona un'opzione.</li> <li>4. Il sistema verifica che il fondo dell'utente sia sufficiente per il costo dell'abbonamento.</li> <li>5. Il sistema addebita il costo e attiva l'abbonamento</li> </ol>
<b>Svolgimenti alternativi</b>	<ul style="list-style-type: none"> <li>• 3a. Se è già presente un abbonamento attivo l'utente è notificato con un messaggio.</li> <li>• 4a. Se il fondo è insufficiente Il sistema mostra un errore e propone di ricaricare il fondo (Test 28).</li> </ul>
<b>Postconditions</b>	Lo stato dell'abbonamento è aggiornato nel database, i benefici (sconti e giochi gratuiti) sono applicati.

Tabella 4: Template che descrive il caso d'uso per la gestione degli abbonamenti.

## 2.3 Mock-Ups

In questa sezione vengono riportate delle possibili tabelle di mockup per il progetto.

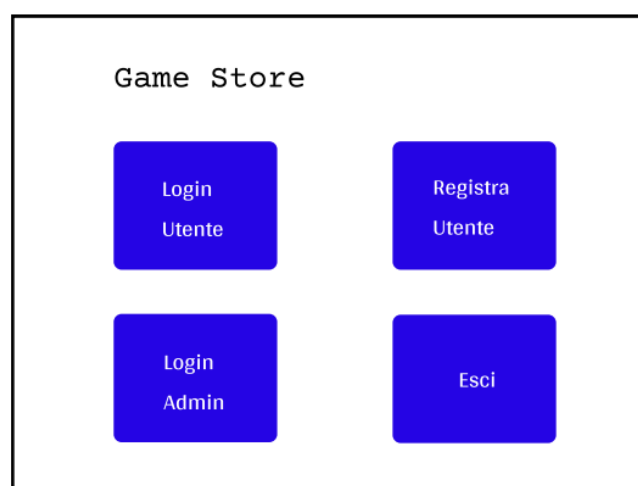


Figura 3: Mockup raffigurante un prototipo del menu iniziale dell'applicazione

The mockup shows a login interface with the title "Login Utente". It contains two input fields: "Username:" and "Password:". Below the password field is a red button labeled "Accedi".

Figura 4: Mockup raffigurante un prototipo del menu di login

The mockup shows a dashboard titled "Benvenuto". It features six blue buttons arranged in a 2x3 grid. The top row contains "Catalogo Giochi", "Carrello", and "Libreria". The bottom row contains "Ricarica Fondo", "Gestione Abbonamento", and "Logout".

Figura 5: Mockup raffigurante un prototipo della schermata principale per l'utente

The mockup shows a catalog titled "Catalogo Giochi". It has three filter buttons: "Genere", "Piattaforma", and "Data", followed by a red "Cerca" button. Below the filters, it says "Risultati:". There are two blue buttons for "FIFA" and "Minecraft", each with a corresponding "Agg. al Carrello" button. At the bottom is a blue button labeled "Torna al Menu".

Figura 6: Mockup raffigurante un prototipo della visualizzazione del catalogo di videogiochi

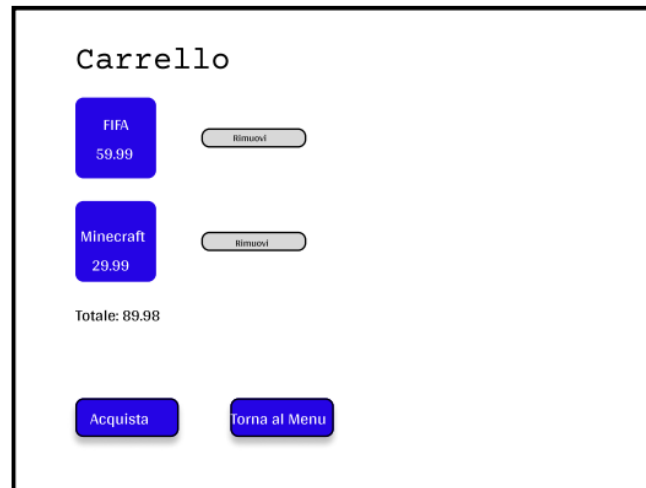


Figura 7: Mockup raffigurante un prototipo della visualizzazione del carrello

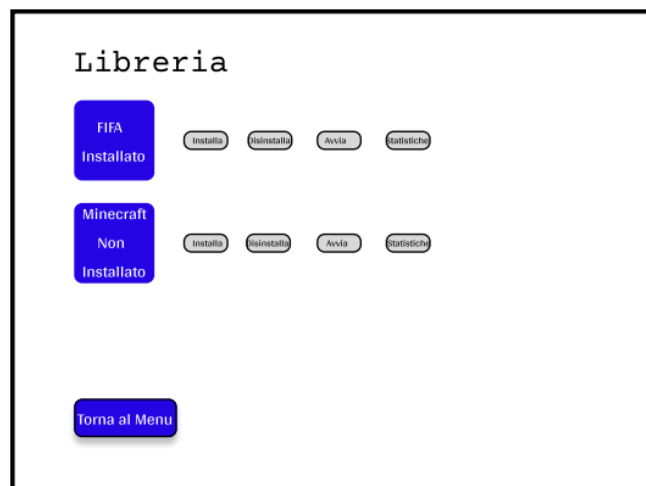


Figura 8: Mockup raffigurante un prototipo della visualizzazione della libreria

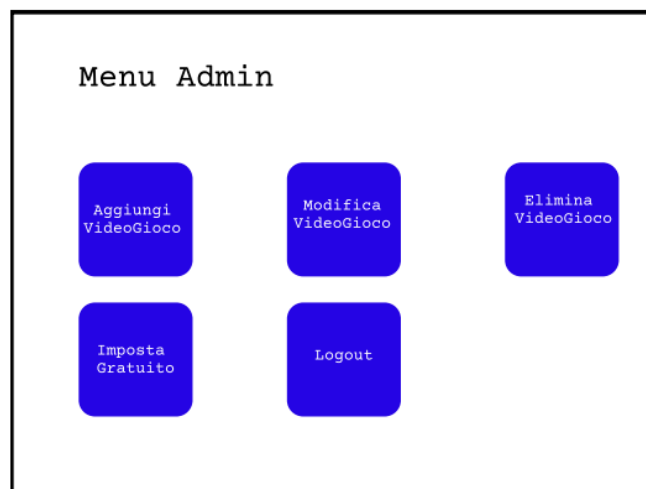


Figura 9: Mockup raffigurante un prototipo del menu dell'admin

## 2.4 Class Diagram

In questa sezione viene riportato il diagramma di classe del progetto

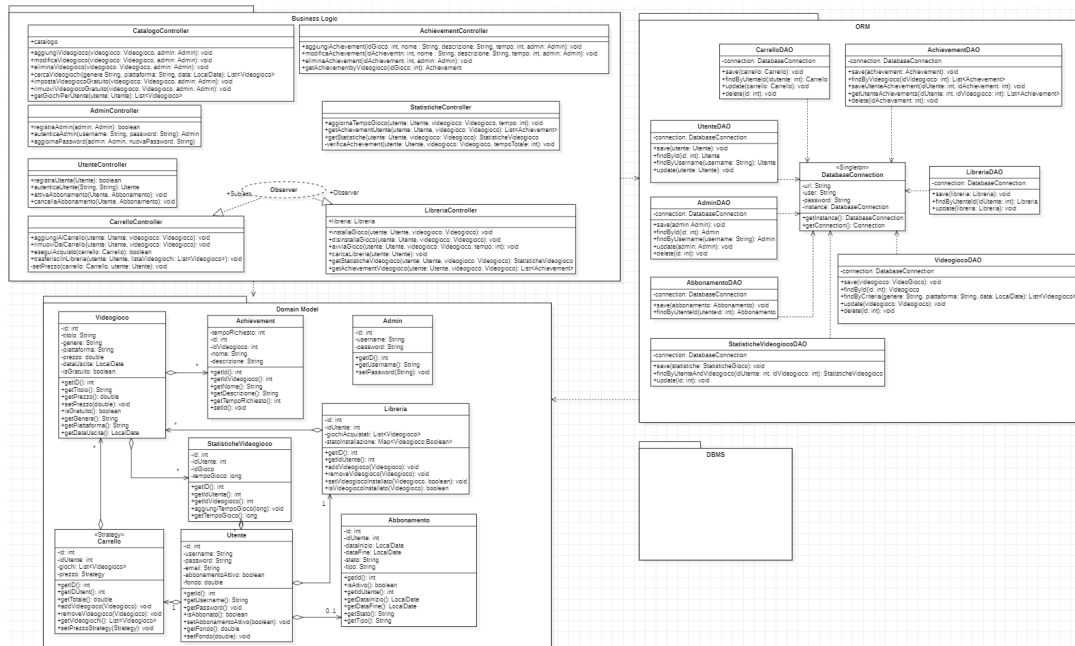


Figura 10: Diagramma di Classe, con suddivisione nei pacchetti Domain Model, Business Logic, ORM

## 2.5 ER Diagram

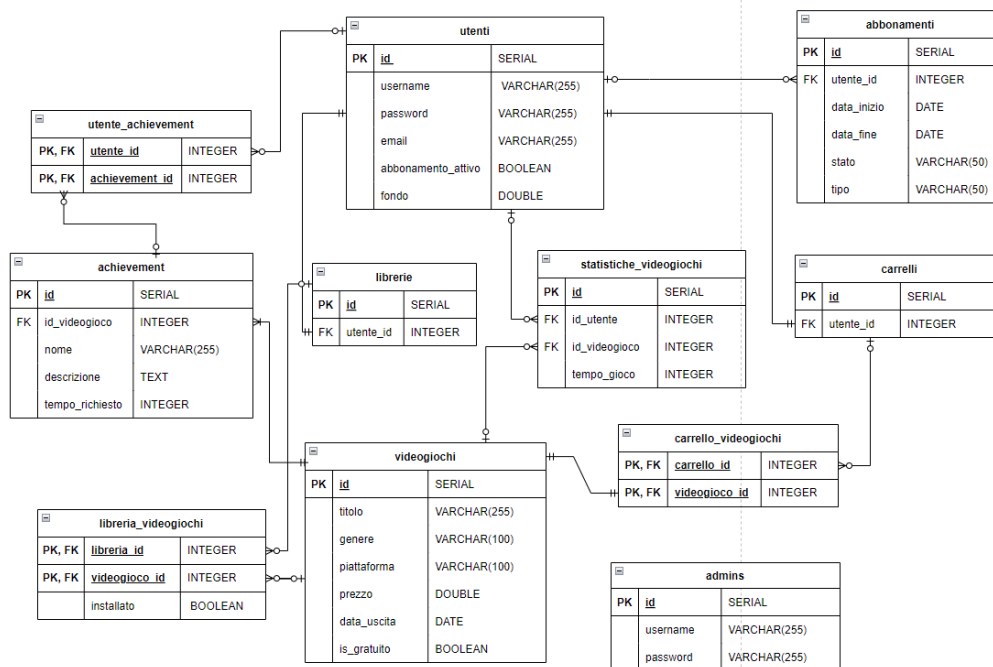


Figura 11: Diagramma ER: Rappresentazione della struttura delle tabelle create dal DBMS

## 3 Implementazione

### 3.1 Domain Model

In questa sezione viene descritto il Domain Model del sistema, che rappresenta le entità principali e le loro interazioni all'interno della piattaforma. Le entità rappresentate sono: gli utenti, i videogiochi, il carrello, la libreria personale, gli abbonamenti, le statistiche, gli achievement e l'admin. Di seguito viene fornita una descrizione dettagliata delle classi principali, con particolare attenzione al loro ruolo, agli attributi e metodi più rilevanti, e alle relazioni con altre classi.

#### 3.1.1 Utente

La classe `Utente` modella un utente registrato nel sistema, con attributi che permettono l'autenticazione e la registrazione (username e password e email), l'acquisto di videogiochi e la gestione degli abbonamenti (abbonamentoAttivo e fondo). In particolare l'attributo `abbonamentoAttivo` determina lo stato di un abbonamento valido associato all'utente, mentre `fondo` determina il saldo associato all'utente. La classe mette inoltre a disposizione metodi getter e setter per gestire gli attributi.

#### 3.1.2 Videogioco

La classe `Videogioco` modella un videogioco disponibile nel catalogo del sistema, con attributi che permettono la gestione del catalogo (`id`, `titolo`, `genere`, `piattaforma`, `dataUscita`) e l'acquisto di videogiochi (`prezzo`, `isGratuito`). In particolare, l'attributo `prezzo` determina il costo del gioco, mentre `isGratuito` indica se il gioco è accessibile gratuitamente tramite un abbonamento.

#### 3.1.3 Carrello

La classe `Carrello` modella il carrello di un utente, con attributi che permettono la selezione di videogiochi per l'acquisto (`videogiochi`, `prezzoStrategy`) e l'associazione all'utente (`idUtente`). In particolare, l'attributo `videogiochi` contiene la lista dei giochi selezionati, mentre `prezzoStrategy` implementa il pattern Strategy tramite l'interfaccia `PrezzoStrategy` e le sue implementazioni (`GoldAbbonatoPrezzoStrategy`, `SilverAbbonatoPrezzoStrategy`, `NonAbbonatoPrezzoStrategy`). Questo pattern consente di calcolare dinamicamente il prezzo dei videogiochi in base allo stato dell'abbonamento dell'utente, applicando uno sconto del 20% per gli utenti Gold, del 10% per gli utenti Silver, o nessun sconto per gli utenti non abbonati.

```
public interface PrezzoStrategy {
    double calculatePrice(Videogioco videogioco, Utente utente);
}
```

Snippet 1: Interfaccia `PrezzoStrategy` per la gestione dinamica dei prezzi

```
public double getTotale(Utente utente) {
    return videogiochi.stream()
        .mapToDouble(videogioco -> prezzoStrategy.calculatePrice(videogioco,
            utente))
        .sum();
}
```

Snippet 2: Metodo per il calcolo del totale nella classe `Carrello`

Lo snippet 1 definisce l'interfaccia `PrezzoStrategy`, che specifica il metodo `calculatePrice()` per calcolare il prezzo di un videogioco, implementata con la rispettiva logica nelle classi `GoldAbbonatoPrezzoStrategy`, `SilverAbbonatoPrezzoStrategy` e `NonAbbonatoPrezzoStrategy`. Lo snippet 2 mostra il metodo `getTotale()`, che somma i prezzi calcolati in base alla strategia di prezzo corrente per tutti i giochi nel carrello. Questi snippet evidenziano come il pattern Strategy consenta di adattare dinamicamente il calcolo dei prezzi in base al tipo di abbonamento dell'utente, garantendo flessibilità e scalabilità.

#### 3.1.4 Libreria

La classe `Libreria` modella la collezione di videogiochi acquistati da un utente, con attributi che permettono la gestione dei giochi posseduti (`videogiochiAcquistati`, `statoInstallazione`) e l'associazione all'utente (`idUtente`). In particolare, l'attributo `videogiochiAcquistati` contiene i giochi acquistati, mentre

`statoInstallazione` traccia lo stato di installazione di ciascun gioco. La classe mette a disposizione metodi getter e setter per gestire gli attributi.

### 3.1.5 Abbonamento

La classe `Abbonamento` modella un abbonamento sottoscritto da un utente, con attributi che permettono la gestione dello stato e della durata dell'abbonamento (`stato`, `tipo`, `dataInizio`, `dataFine`) e l'associazione all'utente (`idUtente`). In particolare, l'attributo `stato` indica se l'abbonamento è attivo, mentre `tipo` specifica se è Gold o Silver, influenzando sconti e benefici. La classe mette a disposizione metodi getter per gestire gli attributi.

### 3.1.6 StatisticheVideogioco

La classe `StatisticheVideogioco` modella le statistiche di gioco di un utente per un videogioco specifico, con attributi che permettono il tracciamento del tempo di gioco e l'associazione all'utente e al videogioco. In particolare, l'attributo `tempoGioco` registra il tempo trascorso giocando, mentre `idUtente` e `idVideogioco` collegano le statistiche al rispettivo utente e gioco. La classe mette a disposizione metodi getter e setter per gestire gli attributi.

### 3.1.7 Achievement

La classe `Achievement` modella un obiettivo ottenibile in un videogioco, con attributi che permettono la descrizione dell'obiettivo (`nome`, `descrizione`, `tempoRichiesto`) e l'associazione al videogioco (`idVideogioco`). In particolare, l'attributo `tempoRichiesto` indica il tempo di gioco necessario per sbloccare l'obiettivo, mentre `nome` e `descrizione` ne definiscono le caratteristiche. La classe mette a disposizione metodi getter e setter per gestire gli attributi.

### 3.1.8 Admin

La classe `Admin` modella un amministratore del sistema, con attributi che permettono l'autenticazione e la gestione del sistema (`username`, `password`) e l'identificazione (`id`). In particolare, l'attributo `username` e `password` garantiscono l'accesso sicuro per le operazioni amministrative, come la gestione del catalogo. La classe mette a disposizione metodi getter e setter per gestire gli attributi.

## 3.2 Business Logic

In questa sezione viene descritto il pacchetto Business Logic del sistema, in cui sono definite le funzionalità operative della piattaforma per la vendita di videogiochi. La Business Logic coordina le interazioni tra le entità del Domain Model (es. `Utente`, `Videogioco`, `Carrello`, `Libreria`, `Abbonamento`, `StatisticheVideogioco`, `Achievement`, `Admin`) e il database tramite i Data Access Object (DAO). Le classi controller implementano le operazioni definite negli use case (si vedano le Tabelle 1, 2, 3, 4), come registrazione e autenticazione degli utenti, gestione del carrello e degli acquisti, attivazione degli abbonamenti, monitoraggio delle statistiche di gioco e gestione amministrativa del catalogo.

Di seguito, vengono descritte le classi principali della Business Logic, con particolare attenzione al loro ruolo, alle interazioni con i DAO e alle funzionalità chiave di ciascuna.

### 3.2.1 UtenteController

La classe `UtenteController` gestisce le operazioni relative agli utenti, come registrazione, autenticazione, attivazione degli abbonamenti e ricarica del saldo, con attributi che permettono l'interazione con il database (`utenteDAO`, `abbonamentoDAO`). In particolare, l'attributo `utenteDAO` gestisce la persistenza degli utenti, mentre `abbonamentoDAO` gestisce gli abbonamenti.

```
public void attivaAbbonamento(Utente utente, Abbonamento abbonamento) {
    double costoAbbonamento = getCostoAbbonamento(abbonamento.getTipo());
    if (utente.getFondo() >= costoAbbonamento) {
        utente.setFondo(utente.getFondo() - costoAbbonamento);
        utente.setAbbonamentoAttivo(true);
        abbonamentoDAO.save(abbonamento);
        utenteDAO.update(utente);
    } else {
```

```

        throw new IllegalStateException("Fondo insufficiente per attivare l'
            abbonamento " + abbonamento.getTipo());
    }
}

```

Snippet 3: Metodo per l'attivazione di un abbonamento nella classe UtenteController

```

public void ricaricaFondo(Utente utente, double importo) {
    if (importo <= 0) {
        throw new IllegalArgumentException("L'importo deve essere positivo");
    }
    utente.setFondo(utente.getFondo() + importo);
    utenteDAO.update(utente);
}

```

Snippet 4: Metodo per la ricarica del saldo nella classe UtenteController

Lo snippet di codice 3 mostra il metodo `attivaAbbonamento()`, che implementa l'attivazione di un abbonamento: verifica il saldo disponibile, addebita il costo, aggiorna lo stato dell'abbonamento e persiste i dati tramite i DAO. Lo snippet 4, invece mostra il metodo `ricaricaFondo()`: verifica se l'importo è valido, aggiorna il fondo in maniera opportuna, salva le modifiche tramite il DAO.

### 3.2.2 CarrelloController

La classe `CarrelloController` gestisce le operazioni relative al carrello di un utente, come l'aggiunta e la rimozione di videogiochi, l'esecuzione degli acquisti e l'aggiornamento della libreria, con attributi che permettono l'interazione con il database (`carrelloDAO`, `utenteDAO`, `abbonamentoDAO`, `libreriaDAO`) e la notifica di eventi (`observers`). Questa svolge il ruolo di `Subject`, in quanto notifica il proprio cambiamento di Stato alla libreria dell'utente nel momento in cui viene completato con successo un acquisto e si deve aggiornare la libreria. La classe utilizza il *pattern Strategy* tramite `PrezzoStrategy` per calcolare i prezzi in base agli abbonamenti, come descritto nel Domain Model.

```

public boolean eseguiAcquisto(Carrello carrello) {
    Utente utente = utenteDAO.findById(carrello.getIdUtente());
    if (utente == null) {
        throw new IllegalStateException("Utente non trovato.");
    }
    double totale = carrello.getTotale(utente);
    Libreria libreria = libreriaDAO.findById(utente.getId());
    if (libreria != null) {
        for (Videogioco v : carrello.getVideogiochi()) {
            if (libreria.getVideogiochiAcquistati().contains(v)) {
                throw new IllegalArgumentException("Il gioco " + v.getTitolo() + " e
                    ' gia' presente nella tua libreria.");
            }
        }
    }
    if (utente.getFondo() >= totale) {
        utente.setFondo(utente.getFondo() - totale);
        utenteDAO.update(utente);
        trasferisciInLibreria(utente, carrello.getVideogiochi());
        notificaAcquistoCompletato(utente, carrello);
        carrelloDAO.delete(carrello.getId());
        return true;
    } else {
        throw new IllegalStateException("Fondo insufficiente per completare l'
            acquisto.");
    }
}

```

Snippet 5: Metodo per l'esecuzione di un acquisto nella classe CarrelloController

```

private void trasferisciInLibreria(Utente utente, List<Videogioco> videogiochi) {
    Libreria libreria = libreriaDAO.findById(utente.getId());
}

```

```

    if (libreria == null) {
        System.out.println("Creazione nuova libreria per utente: " + utente.
            getId());
        libreria = new Libreria(0, utente.getId());
        for (Videogioco v : videogiochi) {
            System.out.println("Aggiungo videogioco alla nuova libreria: " + v.
                getTitolo() + " (ID: " + v.getId() + ")");
            libreria.addVideogioco(v);
            libreria.setVideogiocoInstallato(v, false);
        }
        libreriaDAO.save(libreria);
    } else {
        System.out.println("Aggiornamento libreria esistente: libreria_id=" +
            libreria.getId());
        for (Videogioco v : videogiochi) {
            if (!libreria.getVideogiochiAcquistati().contains(v)) {
                System.out.println("Aggiungo videogioco alla libreria: " + v.
                    getTitolo() + " (ID: " + v.getId() + ")");
                libreria.addVideogioco(v);
                libreria.setVideogiocoInstallato(v, false);
            } else {
                System.out.println("Videogioco gia' presente nella libreria: " +
                    v.getTitolo() + " (ID: " + v.getId() + ")");
            }
        }
        libreriaDAO.update(libreria);
    }
}

```

Snippet 6: Metodo per trasferire i videogiochi acquistati dal carrello alla libreria nella classe CarrelloController

```

public void notificaAcquistoCompletato(Utente utente, Carrello carrello) {
    for (Observer observer : observers) {
        observer.onAcquistoCompletato(utente, carrello);
    }
}

```

Snippet 7: Metodo per la notifica alla libreria dell'acquisto di giochi dal carrello nella classe CarrelloController

```

private void setPrezzoStrategy(Carrello carrello, Utente utente) {
    Abbonamento abbonamento = abbonamentoDAO.findByUtenteId(utente.getId());
    if (abbonamento != null && abbonamento.isAttivo()) {
        switch (abbonamento.getTipo()) {
            case "Silver":
                carrello.setPrezzoStrategy(new SilverAbbonatoPrezzoStrategy());
                break;
            case "Gold":
                carrello.setPrezzoStrategy(new GoldAbbonatoPrezzoStrategy());
                break;
            default:
                carrello.setPrezzoStrategy(new NonAbbonatoPrezzoStrategy());
        }
    } else {
        carrello.setPrezzoStrategy(new NonAbbonatoPrezzoStrategy());
    }
}

```

Snippet 8: Metodo per impostare la strategia di prezzo nella classe CarrelloController

Il Listato 5 mostra il metodo `eseguiAcquisto()`, che implementa l'acquisto dei videogiochi precedentemente inseriti nel carrello, verificando l'utente, il saldo e la libreria, aggiornando il fondo, trasferendo i giochi e notificando gli observer tramite il *pattern Observer* e al metodo `notificaAcquistoCompletato()` (7) che implementa la notifica a tutti gli observer connessi al carrello. Il Listato 6 mostra il metodo `trasferisciInLibreria()`, che sposta i videogiochi acquistati dal carrello di un utente alla libreria ad

esso associato. Il Listato 8 mostra il metodo `setPrezzoStrategy()`, che configura dinamicamente la strategia di prezzo in base all'abbonamento dell'utente. Questi metodi evidenziano il ruolo centrale della classe nella gestione degli acquisti, integrando il *pattern Strategy* per i prezzi e il *pattern Observer* per le notifiche.

### 3.2.3 LibreriaController

La classe `LibreriaController` gestisce le operazioni relative alla libreria di un utente, come il caricamento della libreria, l'installazione e la disinstallazione di videogiochi, l'avvio dei giochi e l'accesso a statistiche e achievement, con attributi che permettono l'interazione con il database (`libreriaDAO`) e la gestione delle statistiche (`statisticheController`). La classe implementa il *pattern Observer* per ricevere notifiche di acquisto da `CarrelloController`. La classe supporta le funzionalità di gestione della libreria e delle attività di gioco, garantendo l'autorizzazione dell'utente e l'aggiornamento dello stato dei giochi.

```
public void installaVideogioco(Utente utente, Videogioco videogioco) {
    if (utente.getId() == libreria.getIdUtente()) {
        if (libreria.getVideogiochiAcquistati().contains(videogioco)) {
            if (!libreria.isVideogiocoInstallato(videogioco)) {
                System.out.println("Installazione del videogioco: " + videogioco.
                    getTitolo());
                libreria.setVideogiocoInstallato(videogioco, true);
                libreriaDAO.update(libreria);
            } else {
                throw new IllegalStateException("Videogioco gia' installato");
            }
        } else {
            throw new IllegalArgumentException("Videogioco non acquistato");
        }
    } else {
        throw new SecurityException("Utente non autorizzato");
    }
}
```

Snippet 9: Metodo per installare un videogioco nella classe `LibreriaController`

```
public void avviaVideogioco(Utente utente, Videogioco videogioco, int tempo) {
    if (utente.getId() == libreria.getIdUtente()) {
        if (libreria.getVideogiochiAcquistati().contains(videogioco)) {
            if (libreria.isVideogiocoInstallato(videogioco)) {
                System.out.println("Avvio del videogioco: " + videogioco.getTitolo()
                    + " per " + tempo + " secondi");
                statisticheController.aggiornaTempoGioco(utente, videogioco, tempo);
            } else {
                throw new IllegalStateException("Videogioco non installato");
            }
        } else {
            throw new IllegalArgumentException("Videogioco non acquistato");
        }
    } else {
        throw new SecurityException("Utente non autorizzato");
    }
}
```

Snippet 10: Metodo per avviare un videogioco nella classe `LibreriaController`

```
@Override
public void onAcquistoCompletato(Utente utente, Carrello carrello) {
    if (utente.getId() == libreria.getIdUtente()) {
        String titoli = carrello.getVideogiochi().stream()
            .map(Videogioco::getTitolo)
            .collect(Collectors.joining(", "));
        System.out.println("Notifica: Videogiochi acquistati e aggiunti alla
            libreria di " + utente.getUsername() + ": " + titoli);
    } else {
        throw new SecurityException("Utente non autorizzato");
    }
}
```

```

    }
}

```

Snippet 11: Metodo per ricevere notifiche di acquisto nella classe LibreriaController

Lo snippet 9 mostra il metodo `installaVideogioco()`, che installa un videogioco acquistato, verificando l'autorizzazione dell'utente e lo stato del gioco, e aggiorna la libreria. Lo snippet 10 mostra il metodo `avviaVideogioco()`, che avvia un videogioco installato, aggiornando il tempo di gioco tramite `statisticheController`. Lo snippet 11 mostra il metodo `onAcquistoCompletato()`, che implementa il metodo `update`, tipico del *pattern Observer*, e riceve notifiche di acquisto, aggiornando la libreria con i nuovi giochi.

### 3.2.4 CatalogoController

La classe `CatalogoController` mette a disposizione le operazioni relative al catalogo e permette sia la gestione da parte dell'admin dei videogiochi del catalogo (aggiunta, modifica, eliminazione) sia la ricerca dei giochi, con attributi che permettono l'interazione con il database (`videogiocoDAO`, `adminDAO`).

```

public void aggiungiVideogioco(Videogioco videogioco, Admin admin) {
    if (adminDAO.findById(admin.getId()) == null) {
        throw new SecurityException("Admin non autenticato");
    }
    if (videogiocoDAO.findById(videogioco.getId()) == null) {
        videogiocoDAO.save(videogioco);
    } else {
        throw new IllegalArgumentException("Videogioco già esistente");
    }
}

```

Snippet 12: Metodo per aggiungere un videogioco al catalogo nella classe CatalogoController

```

public List<Videogioco> cercaVideogiochi(String genere, String piattaforma,
    LocalDate data) {
    return videogiocoDAO.findByCriteria(genere, piattaforma, data);
}

```

Snippet 13: Metodo per cercare videogiochi nel catalogo nella classe CatalogoController

```

public List<Videogioco> getVideogiochiPerUtente(Utente utente) {
    List<Videogioco> videogiochi = videogiocoDAO.findByCriteria(null, null, null);
    if (utente != null && utente.isAbbonamentoAttivo()) {
        Abbonamento abbonamento = new AbbonamentoDAO().findById(utente.getId());
        PrezzoStrategy prezzoStrategy = switch (abbonamento != null ? abbonamento.
            getTipo() : "Nessuno") {
            case "Silver" -> new SilverAbbonatoPrezzoStrategy();
            case "Gold" -> new GoldAbbonatoPrezzoStrategy();
            default -> new NonAbbonatoPrezzoStrategy();
        };
        return videogiochi.stream()
            .map(v -> new Videogioco(
                v.getId(),
                v.getTitolo(),
                v.getGenere(),
                v.getPiattaforma(),
                prezzoStrategy.calculatePrice(v, utente),
                v.getDataUscita(),
                v.isGratuito()
            ))
            .collect(Collectors.toList());
    }
    return videogiochi;
}

```

Snippet 14: Metodo per ottenere videogiochi personalizzati per un utente nella classe CatalogoController

Lo snippet 12 mostra il metodo `aggiungiVideogioco()`, che consente a un amministratore autenticato di aggiungere un videogioco al catalogo, verificando l'unicità. Lo snippet 13 mostra il metodo `cercaVideogiochi()`, che restituisce una lista di videogiochi filtrata per genere, piattaforma o data, supportando la ricerca degli utenti. Lo snippet 14 mostra il metodo `getVideogiochiPerUtente()`, che restituisce il catalogo con prezzi personalizzati in base all'abbonamento dell'utente. Questi metodi evidenziano il ruolo della classe nella gestione amministrativa e nell'accesso personalizzato al catalogo.

### 3.2.5 StatisticheController

La classe `StatisticheController` gestisce le statistiche di gioco e gli achievement degli utenti, come l'aggiornamento del tempo di gioco e la verifica degli obiettivi sbloccati, con attributi che permettono l'interazione con il database (`statisticheDAO`, `achievementDAO`). La classe supporta le funzionalità di monitoraggio delle attività di gioco, integrandosi con `LibreriaController` per aggiornare il tempo di gioco e sbloccare gli achievement.

```
public void aggiornaTempoGioco(Utente utente, Videogioco videogioco, int tempo) {
    StatisticheVideogioco statistiche = statisticheDAO.findByUtenteAndVideogioco(
        utente.getId(), videogioco.getId());
    if (statistiche == null) {
        statistiche = new StatisticheVideogioco(0, utente.getId(), videogioco.getId(), tempo);
    } else {
        statistiche.setTempoGioco(statistiche.getTempoGioco() + tempo);
    }
    statisticheDAO.save(statistiche);
    verificaAchievement(utente, videogioco, statistiche.getTempoGioco());
}
```

Snippet 15: Metodo per aggiornare il tempo di gioco nella classe `StatisticheController`

```
private void verificaAchievement(Utente utente, Videogioco videogioco, int
tempoTotal) {
    List<Achievement> achievements = achievementDAO.findByVideogioco(videogioco.
getId());
    List<Achievement> utenteAchievements = achievementDAO.getUtenteAchievements(
        utente.getId(), videogioco.getId());
    for (Achievement achievement : achievements) {
        boolean isAlreadyAchieved = utenteAchievements.stream()
            .anyMatch(a -> a.getId() == achievement.getId());
        if (!isAlreadyAchieved && tempoTotal >= achievement.getTempoRichiesto()) {
            achievementDAO.saveUtenteAchievement(utente.getId(), achievement.getId()
            );
        }
    }
}
```

Snippet 16: Metodo per verificare gli achievement nella classe `StatisticheController`

Lo snippet 15 mostra il metodo `aggiornaTempoGioco()`, che aggiorna il tempo di gioco di un utente per un videogioco, creando una nuova statistica se necessario e verificando gli achievement. Lo snippet 16 mostra il metodo `verificaAchievement()`, che controlla se il tempo di gioco soddisfa i requisiti per sbloccare nuovi achievement, salvandoli nel database.

### 3.2.6 AchievementController

La classe `AchievementController` gestisce le operazioni relative agli achievement dei videogiochi, come l'aggiunta, la modifica, l'eliminazione e il recupero degli achievement da parte dell'admin.

```
public void aggiungiAchievement(int idVideogioco, String nome, String descrizione,
int tempoRichiesto, Admin admin) {
    if (adminDAO.findById(admin.getId()) == null) {
        throw new SecurityException("Admin non autenticato");
    }
}
```

```

Achievement achievement = new Achievement(0, idVideogioco, nome, descrizione,
    tempoRichiesto);
achievementDAO.save(achievement);
}

```

Snippet 17: Metodo per aggiungere un achievement nella classe AchievementController

Lo snippet 17 mostra il metodo `aggiungiAchievement()`, che consente a un amministratore autenticato di aggiungere un nuovo achievement per un videogioco, persistendolo nel database.

### 3.2.7 AdminController

La classe `AdminController` gestisce le operazioni relative agli amministratori, come registrazione, autenticazione e aggiornamento della password, con un attributo che permette l'interazione con il database (`adminDAO`).

```

public boolean registraAdmin(Admin admin) {
    if (adminDAO.findByUsername(admin.getUsername()) != null) {
        return false;
    }
    adminDAO.save(admin);
    return true;
}

```

Snippet 18: Metodo per registrare un amministratore nella classe AdminController

Lo snippet 18 mostra il metodo `registraAdmin()`, che registra un nuovo amministratore, verificando l'unicità dell'username e persistendolo nel database.

## 3.3 ORM (Object-Relational Mapping)

Questa sezione descrive il livello di accesso ai dati del sistema, implementato tramite Object-Relational Mapping (ORM) per gestire la persistenza delle entità del Domain Model (`Utente`, `Videogioco`, `Carrello`, `Libreria`, `Abbonamento`, `StatisticheVideogioco`, `Achievement`, `Admin`) su un database PostgreSQL. La classe `DatabaseConnection` fornisce una connessione Singleton al DBMS, mentre le classi DAO (Data Access Object) gestiscono le operazioni CRUD (Create, Read, Update, Delete) per ogni entità.

### 3.3.1 DatabaseConnection

La classe `DatabaseConnection` implementa il pattern Singleton per fornire un'unica istanza di connessione al database PostgreSQL, utilizzando le credenziali definite (URL, username, password). L'attributo `instance` garantisce l'unicità della connessione, mentre il metodo `getConnection()` restituisce un oggetto `Connection` per le operazioni dei DAO.

```

public Connection getConnection() throws SQLException {
    return DriverManager.getConnection(URL, USER, PASSWORD);
}

```

Snippet 19: Metodo per ottenere la connessione al database nella classe DatabaseConnection

Lo snippet 19 mostra il metodo `getConnection()`, che stabilisce la connessione al database PostgreSQL, utilizzata da tutti i DAO per eseguire query.

### 3.3.2 UtenteDAO

La classe `UtenteDAO` gestisce la persistenza degli oggetti `Utente`, supportando operazioni CRUD per registrazione, autenticazione e aggiornamento del profilo. L'attributo `connection` consente l'interazione con il database tramite `DatabaseConnection`.

```

public Utente save(Utente utente) {
    String sql = "INSERT INTO utenti (username, password, email, abbonamento_attivo,
        fondo) VALUES (?, ?, ?, ?, ?) RETURNING id";
    try (PreparedStatement stmt = connection.prepareStatement(sql)) {
        stmt.setString(1, utente.getUsername());
        stmt.setString(2, utente.getPassword());
        stmt.setString(3, utente.getEmail());
    }
}

```

```

stmt.setBoolean(4, utente.isAbbonamentoAttivo());
stmt.setDouble(5, utente.getFondo());
ResultSet rs = stmt.executeQuery();
if (rs.next()) {
    int generatedId = rs.getInt("id");
    return new Utente(
        generatedId,
        utente.getUsername(),
        utente.getPassword(),
        utente.getEmail(),
        utente.isAbbonamentoAttivo(),
        utente.getFondo()
    );
}
throw new SQLException("Inserimento fallito, nessun ID generato.");
} catch (SQLException e) {
    throw new RuntimeException("Errore durante il salvataggio dell'utente", e);
}
}

```

Snippet 20: Metodo per salvare un utente nella classe UtenteDAO

Lo snippet 20 mostra il metodo `save()`, che inserisce un nuovo utente nel database, restituendo un oggetto `Utente` con l'ID generato, supportando la registrazione. Altri metodi includono `update()` per aggiornare i dati utente, `findById()` e `findByUsername()` per il recupero.

### 3.3.3 VideogiocoDAO

La classe `VideogiocoDAO` gestisce la persistenza degli oggetti `Videogioco`, supportando operazioni CRUD per la gestione del catalogo. L'attributo `connection` consente l'interazione con il database tramite `DatabaseConnection`.

```

public List<Videogioco> findByCriteria(String genere, String piattaforma, LocalDate
data) {
    StringBuilder sql = new StringBuilder("SELECT * FROM videogiochi WHERE 1=1");
    List<Object> params = new ArrayList<>();
    if (genere != null) {
        sql.append(" AND genere = ?");
        params.add(genere);
    }
    if (piattaforma != null) {
        sql.append(" AND piattaforma = ?");
        params.add(piattaforma);
    }
    if (data != null) {
        sql.append(" AND data_uscita = ?");
        params.add(Date.valueOf(data));
    }
    try (PreparedStatement stmt = connection.prepareStatement(sql.toString())) {
        for (int i = 0; i < params.size(); i++) {
            stmt.setObject(i + 1, params.get(i));
        }
        try (ResultSet rs = stmt.executeQuery()) {
            List<Videogioco> videogiochi = new ArrayList<>();
            while (rs.next()) {
                videogiochi.add(new Videogioco(
                    rs.getInt("id"),
                    rs.getString("titolo"),
                    rs.getString("genere"),
                    rs.getString("piattaforma"),
                    rs.getDouble("prezzo"),
                    rs.getDate("data_uscita") != null ? rs.getDate("data_uscita").
                        toLocalDate() : null,
                    rs.getBoolean("is_gratuito")
                ));
            }
        }
    }
}

```

```

        ));
    }
    return videogiochi;
}
} catch (SQLException e) {
    throw new RuntimeException("Errore durante la ricerca dei videogiochi", e);
}
}

```

Snippet 21: Metodo per cercare videogiochi nella classe VideogiocoDAO

Lo snippet 21 mostra il metodo `findByCriteria()`, che recupera videogiochi filtrati per genere, piattaforma o data, supportando la ricerca nel catalogo. Altri metodi includono `save()` per aggiungere videogiochi, `update()` per modificarli e `delete()` per eliminarli.

### 3.3.4 CarrelloDAO

La classe `CarrelloDAO` gestisce la persistenza degli oggetti `Carrello`, supportando operazioni CRUD per la gestione del carrello utente. L'attributo `connection` consente l'interazione con il database tramite `DatabaseConnection`.

```

public void save(Carrello carrello) {
    String sql = "INSERT INTO carrelli (id, utente_id) VALUES (?, ?) ON CONFLICT (id) DO UPDATE SET utente_id = EXCLUDED.utente_id";
    try (PreparedStatement stmt = connection.prepareStatement(sql)) {
        stmt.setInt(1, carrello.getId());
        stmt.setInt(2, carrello.getIdUtente());
        stmt.executeUpdate();
        sql = "DELETE FROM carrello_videogiochi WHERE carrello_id = ?";
        try (PreparedStatement deleteStmt = connection.prepareStatement(sql)) {
            deleteStmt.setInt(1, carrello.getId());
            deleteStmt.executeUpdate();
        }
        for (Videogioco videoggioco : carrello.getVideogiochi()) {
            sql = "INSERT INTO carrello_videogiochi (carrello_id, videoggioco_id) VALUES (?, ?)";
            try (PreparedStatement videoStmt = connection.prepareStatement(sql)) {
                videoStmt.setInt(1, carrello.getId());
                videoStmt.setInt(2, videoggioco.getId());
                videoStmt.executeUpdate();
            }
        }
    } catch (SQLException e) {
        throw new RuntimeException("Errore durante il salvataggio del carrello", e);
    }
}

```

Snippet 22: Metodo per salvare un carrello nella classe `CarrelloDAO`

Lo snippet 22 mostra il metodo `save()`, che salva o aggiorna un carrello e i suoi videogiochi associati, gestendo la relazione nella tabella `carrello_videogiochi`. Altri metodi includono `update()` per aggiornare il carrello, `findByUtenteId()` per recuperarlo e `delete()` per eliminarlo.

### 3.3.5 LibreriaDAO

La classe `LibreriaDAO` gestisce la persistenza degli oggetti `Libreria`, supportando operazioni CRUD per la gestione dei videogiochi acquistati e installati. L'attributo `connection` consente l'interazione con il database tramite `DatabaseConnection`.

```

public void save(Libreria libreria) {
    String insertLibreria = "INSERT INTO librerie (utente_id) VALUES (?) RETURNING id";
    try (PreparedStatement stmt = connection.prepareStatement(insertLibreria)) {
        stmt.setInt(1, libreria.getIdUtente());
        try (ResultSet rs = stmt.executeQuery()) {

```

```

        if (rs.next()) {
            libreria.setId(rs.getInt("id"));
        }
    }
    String insertVideogioco = "INSERT INTO libreria_videogiochi (libreria_id,
        videogioco_id, installato) VALUES (?, ?, ?) " +
        "ON CONFLICT (libreria_id, videogioco_id) DO UPDATE SET installato =
            EXCLUDED.installato";
    try (PreparedStatement videoStmt = connection.prepareStatement(
        insertVideogioco)) {
        for (Videogioco v : libreria.getVideogiochiAcquistati()) {
            videoStmt.setInt(1, libreria.getId());
            videoStmt.setInt(2, v.getId());
            videoStmt.setBoolean(3, libreria.isVideogiocoInstallato(v));
            videoStmt.addBatch();
        }
        videoStmt.executeBatch();
    }
} catch (SQLException e) {
    throw new RuntimeException("Errore durante il salvataggio della libreria", e
    );
}
}

```

Snippet 23: Metodo per salvare una libreria nella classe LibreriaDAO

Lo snippet 23 mostra il metodo `save()`, che salva una libreria e i suoi videogiochi associati, gestendo lo stato di installazione. Altri metodi includono `findByUtenteId()` per recuperare la libreria e `update()` per modificarla.

### 3.3.6 AbbonamentoDAO

La classe `AbbonamentoDAO` gestisce la persistenza degli oggetti `Abbonamento`, supportando operazioni CRUD per la gestione degli abbonamenti utente. L'attributo `connection` consente l'interazione con il database tramite `DatabaseConnection`.

```

public void save(Abbonamento abbonamento) {
    String sql = ""
        INSERT INTO abbonamenti (id, utente_id, data_inizio, data_fine, stato, tipo)
        VALUES (?, ?, ?, ?, ?, ?)
        ON CONFLICT (id) DO UPDATE
        SET utente_id = EXCLUDED.utente_id,
            data_inizio = EXCLUDED.data_inizio,
            data_fine = EXCLUDED.data_fine,
            stato = EXCLUDED.stato,
            tipo = EXCLUDED.tipo
    "";
    try (PreparedStatement stmt = connection.prepareStatement(sql)) {
        stmt.setInt(1, abbonamento.getId());
        stmt.setInt(2, abbonamento.getIdUtente());
        stmt.setDate(3, Date.valueOf(abbonamento.getDataInizio()));
        stmt.setDate(4, Date.valueOf(abbonamento.getDataFine()));
        stmt.setString(5, abbonamento.getStato());
        stmt.setString(6, abbonamento.getTipo());
        stmt.executeUpdate();
    } catch (SQLException e) {
        throw new RuntimeException("Errore durante il salvataggio dell'abbonamento",
            e);
    }
}

```

Snippet 24: Metodo per salvare un abbonamento nella classe `AbbonamentoDAO`

Lo snippet 24 mostra il metodo `save()`, che salva o aggiorna un abbonamento, gestendo i dati relativi a stato e tipo. Altri metodi includono `findByUtenteId()` per recuperare l'abbonamento.

### 3.3.7 StatisticheVideogiocoDAO

La classe `StatisticheVideogiocoDAO` gestisce la persistenza degli oggetti `StatisticheVideogioco`, supportando operazioni CRUD per il monitoraggio del tempo di gioco. L'attributo `connection` consente l'interazione con il database tramite `DatabaseConnection`.

```
public void save(StatisticheVideogioco statistiche) {
    String sql = "INSERT INTO statistiche_videogiochi (id_utente, id_videogioco,
        tempo_gioco) VALUES (?, ?, ?) " +
        "ON CONFLICT (id_utente, id_videogioco) DO UPDATE SET tempo_gioco =
        EXCLUDED.tempo_gioco";
    try (PreparedStatement stmt = connection.prepareStatement(sql)) {
        stmt.setInt(1, statistiche.getIdUtente());
        stmt.setInt(2, statistiche.getIdVideogioco());
        stmt.setInt(3, statistiche.getTempoGioco());
        stmt.executeUpdate();
    } catch (SQLException e) {
        throw new RuntimeException("Errore nel salvataggio delle statistiche", e);
    }
}
```

Snippet 25: Metodo per salvare statistiche di gioco nella classe `StatisticheVideogiocoDAO`

Lo snippet 25 mostra il metodo `save()`, che salva o aggiorna le statistiche di gioco, gestendo il tempo di gioco. Altri metodi includono `findByUtenteAndVideogioco()` per il recupero e `delete()` per l'eliminazione.

### 3.3.8 AchievementDAO

La classe `AchievementDAO` gestisce la persistenza degli oggetti `Achievement`, supportando operazioni CRUD per la gestione degli achievement dei videogiochi e il loro assegnamento agli utenti. L'attributo `connection` consente l'interazione con il database tramite `DatabaseConnection`.

```
public List<Achievement> getUtenteAchievements(int idUtente, int idVideogioco) {
    List<Achievement> achievements = new ArrayList<>();
    String sql = ""
        SELECT a.*
        FROM achievement a
        JOIN utente_achievement ua ON a.id = ua.id_achievement
        WHERE ua.id_utente = ? AND a.id_videogioco = ?
        "";
    try (PreparedStatement stmt = connection.prepareStatement(sql)) {
        stmt.setInt(1, idUtente);
        stmt.setInt(2, idVideogioco);
        try (ResultSet rs = stmt.executeQuery()) {
            while (rs.next()) {
                achievements.add(new Achievement(
                    rs.getInt("id"),
                    rs.getInt("id_videogioco"),
                    rs.getString("nome"),
                    rs.getString("descrizione"),
                    rs.getInt("tempo_richiesto")
                ));
            }
        }
    } catch (SQLException e) {
        throw new RuntimeException("Errore nel recupero degli achievement utente", e);
    }
    return achievements;
}
```

Snippet 26: Metodo per recuperare gli achievement di un utente nella classe `AchievementDAO`

Lo snippet 26 mostra il metodo `getUtenteAchievements()`, che recupera gli achievement sbloccati da un utente per un videogioco, utilizzando una join con la tabella `utente_achievement`. Altri metodi

includono `save()`, `findByVideogioco()`, `saveUtenteAchievement()` e `delete()` per gestione e assegnamento.

### 3.3.9 AdminDAO

La classe `AdminDAO` gestisce la persistenza degli oggetti `Admin`, supportando operazioni CRUD per la gestione degli amministratori. L'attributo `connection` consente l'interazione con il database tramite `DatabaseConnection`.

```
public void save(Admin admin) {
    String sql = "INSERT INTO admins (username, password) VALUES (?, ?) RETURNING id";
    try (PreparedStatement stmt = connection.prepareStatement(sql)) {
        stmt.setString(1, admin.getUsername());
        stmt.setString(2, admin.getPassword());
        try (ResultSet rs = stmt.executeQuery()) {
            if (rs.next()) {
                admin.setId(rs.getInt("id"));
            }
        }
    } catch (SQLException e) {
        throw new RuntimeException("Errore nel salvataggio dell'admin", e);
    }
}
```

Snippet 27: Metodo per salvare un amministratore nella classe `AdminDAO`

Lo snippet 27 mostra il metodo `save()`, che salva un amministratore e imposta l'ID generato grazie al setter della classe `Admin`. Altri metodi includono `findById()`, `findByUsername()`, `update()` e `delete()` per gestione e autenticazione.

## 3.4 Database

Il database utilizzato per il progetto è implementato in PostgreSQL e interfacciato tramite oggetti DAO, che garantiscono un'astrazione tra la logica di business e l'accesso ai dati. Per assicurare uno stato coerente del database durante l'esecuzione dei test, sono stati creati tre file SQL specifici:

- `reset.sql`: elimina tutte le tabelle del database tramite comandi `DROP TABLE`, garantendo un ambiente pulito prima di ogni test.
- `ProgettoSWE.sql`: definisce lo schema del database, creando tutte le tabelle necessarie con i relativi vincoli e strutture.
- `InserimentoProgettoSWE.sql`: popola le tabelle con dati iniziali, consentendo di inizializzare il database con uno stato predefinito.

Per mantenere la coerenza durante i test, i file vengono eseguiti in sequenza (`reset.sql`, `ProgettoSWE.sql`, `InserimentoProgettoSWE.sql`) all'interno delle funzioni `beforeEach` e `afterEach`, garantendo che ogni test operi su un database nello stesso stato iniziale e che le modifiche effettuate durante i test vengano eliminate al termine.

## 4 Test

### 4.1 Business Logic Test

In questa sezione vengono presentati i test sui metodi esposti dalle classi del pacchetto Business Logic. Essi mirano a verificare il comportamento della piattaforma rispetto ai requisiti funzionali, senza fare assunzioni sulla struttura interna delle classi.

L'obiettivo principale dei test è stato quello di validare il flusso operativo standard, come l'aggiunta di un videogioco al carrello, l'esecuzione di un acquisto, o l'installazione di un videogioco nella libreria di un utente. Tuttavia, particolare attenzione è stata posta sui casi di fallimento e sui comportamenti anomali, come tentativi di autenticazione con credenziali errate, installazione di videogiochi non acquistati, o disinstallazione di videogiochi non installati. Questi scenari sono stati testati per garantire che il sistema gestisca correttamente le eccezioni e fornisca feedback appropriati, mantenendo la robustezza e l'affidabilità della piattaforma.

I test sono stati implementati utilizzando il framework JUnit 5, con l'inizializzazione del database tramite script SQL (`reset.sql`, `ProgettoSWE.sql`, `InserimentoProgettoSWE.sql`), che ha permesso di caricare dati predefiniti per simulare scenari realistici senza creare nuovi oggetti, in linea con i requisiti della piattaforma.

#### 4.1.1 UtenteControllerTest

I test in `UtenteControllerTest` verificano il comportamento delle operazioni legate agli utenti, come la registrazione, l'autenticazione, la ricarica del fondo e l'attivazione degli abbonamenti, rispetto ai requisiti funzionali.

I test implementati in `UtenteControllerTest` coprono i seguenti scenari:

- **testRegistraUtente\_Successo:** Verifica la registrazione di un nuovo utente con credenziali uniche. Si aspetta che il metodo `registraUtente` restituisca un oggetto `Utente` non nullo con lo stesso username fornito (es. "nuovoUser").
- **testRegistraUtente\_UsernameEsistente:** Testa il tentativo di registrazione con un username già esistente (es. "mario"). Si aspetta che venga lanciata un'`IllegalArgumentException`.
- **testAutenticaUtente\_CredenzialiValide:** Valida l'autenticazione di un utente con credenziali corrette (es. "mario", "password123"). Si aspetta che venga restituito un oggetto `Utente` non nullo.
- **testRicaricaFondo\_Successo:** Verifica la ricarica del fondo di un utente (es. "mario") con un importo valido (es. 30.0). Si aspetta che il fondo aggiornato sia incrementato correttamente (es. fondo iniziale + 30.0).
- **testAttivaAbbonamento\_Successo:** Verifica l'attivazione di un abbonamento `Silver` per un utente con fondi sufficienti (es. "luigi"). Si aspetta che l'abbonamento sia attivo e che il fondo venga ridotto di 9.99.
- **testAttivaAbbonamento\_Fallimento\_FondiInsufficienti:** Testa l'attivazione di un abbonamento `Silver` per un utente con fondi insufficienti (es. "toad", fondo 0.0). Si aspetta che venga lanciata un'`IllegalStateException` con il messaggio "Fondo insufficiente per attivare l'abbonamento Silver".

```
@Test
void testRegistraUtente_Successo() {
    Utente nuovo = new Utente(0, "nuovoUser", "pass123", "nuovo@email.com",
        false, 0.0);
    Utente res = utenteController.registraUtente(nuovo);
    assertNotNull(res);
    assertEquals("nuovoUser", res.getUsername());
}

@Test
void testRegistraUtente_UsernameEsistente() {
    Utente u = new Utente(0, "mario", "pwd", "dup@email.com", false, 0.0);
    assertThrows(IllegalArgumentException.class, () -> utenteController.
        registraUtente(u));
}
```

```

@Test
void testAutenticaUtente_CredenzialiValide() {
    Utente res = utenteController.autenticaUtente("mario", "password123");
    assertNotNull(res);
}

@Test
void testRicaricaFondo_Successo() {
    Utente u = utenteController.autenticaUtente("mario", "password123");
    double fondoIniziale = u.getFondo();
    utenteController.ricaricaFondo(u, 30.0);
    Utente aggiornato = utenteController.autenticaUtente("mario", "password123")
        ;
    assertEquals(fondoIniziale + 30.0, aggiornato.getFondo(), 0.01);
}

@Test
void testAttivaAbbonamento_Successo() {
    Utente utente = utenteController.autenticaUtente("luigi", "password456");
    double fondoIniziale = utente.getFondo();
    LocalDate oggi = LocalDate.now();
    Abbonamento abbonamento = new Abbonamento(0, utente.getId(), oggi, oggi.
        plusMonths(1), "attivo", "Silver");
    utenteController.attivaAbbonamento(utente, abbonamento);
    Utente aggiornato = utenteController.autenticaUtente("luigi", "password456")
        ;
    assertTrue(aggiornato.isAbbonamentoAttivo());
    assertEquals(fondoIniziale - 9.99, aggiornato.getFondo(), 0.01);
}

@Test
void testAttivaAbbonamento_Fallimento_FondiInsufficienti() {
    Utente utente = utenteController.autenticaUtente("toad", "password101");
    LocalDate oggi = LocalDate.now();
    Abbonamento abbonamento = new Abbonamento(0, utente.getId(), oggi, oggi.
        plusMonths(1), "attivo", "Silver");
    Exception exception = assertThrows(IllegalStateException.class, () -> {
        utenteController.attivaAbbonamento(utente, abbonamento);
    });
    assertEquals("Fondo insufficiente per attivare l'abbonamento Silver",
        exception.getMessage());
}
}

```

Snippet 28: Test di UtenteController

#### 4.1.2 CatalogoControllerTest

I test in `CatalogoControllerTest` verificano il comportamento delle operazioni legate alla gestione del catalogo di videogiochi, come l'aggiunta, la modifica, l'eliminazione, l'impostazione dello stato gratuito dei videogiochi, da parte dell'admin, e la ricerca.

I test implementati in `CatalogoControllerTest` coprono i seguenti scenari (quelli sulla modifica e l'eliminazione vengono coperti con lo stesso approccio di quelli sull'aggiunta):

- **testAggiungiVideogioco\_Successo:** Verifica l'aggiunta di un nuovo videogioco al catalogo da parte di un amministratore (es. admin). Si aspetta che il metodo `aggiungiVideogioco` inserisca correttamente il videogioco (es. "New Game") e che sia ritrovabile tramite una ricerca per genere (es. "RPG").
- **testAggiungiVideogioco\_Fallimento\_VideogiocoEsistente:** Testa il tentativo di aggiungere un videogioco già esistente (es. "Super Mario Odyssey"). Si aspetta che venga lanciata un'`IllegalArgumentException` con il messaggio "Videogioco già esistente", garantendo che la piattaforma impedisca duplicazioni.

- **testCercaVideogiochi\_Successo:** Verifica la ricerca di videogiochi per genere (es. "Platform"). Si aspetta che il metodo `cercaVideogiochi` restituisca una lista non vuota contenente i videogiochi corrispondenti (es. "Super Mario Odyssey").
- **testCercaVideogiochi\_Fallimento\_NessunRisultato:** Testa la ricerca di videogiochi con un genere inesistente (es. "NonEsistente"). Si aspetta che il metodo `cercaVideogiochi` restituisca una lista vuota, indicando l'assenza di risultati.

```
@Test
void testAggiungiVideogioco_Successo() {
    Admin admin = new Admin(1, "admin", "admin123");
    Videogioco videogioco = new Videogioco(0, "New Game", "RPG", "PC", 29.99,
        LocalDate.now(), false);
    catalogoController.aggiungiVideogioco(videogioco, admin);
    List<Videogioco> videogiochi = catalogoController.cercaVideogiochi("RPG", null,
        null);
    assertFalse(videogiochi.isEmpty());
    assertTrue(videogiochi.stream().anyMatch(v -> v.getTitolo().equals("New Game")))
        ;
}

@Test
void testAggiungiVideogioco_Fallimento_VideogiocoEsistente() {
    Admin admin = new Admin(1, "admin", "admin123");
    Videogioco videogioco = new Videogioco(1, "Super Mario Odyssey", "Platform", "
        Nintendo Switch", 59.99, LocalDate.parse("2017-10-27"), false);
    Exception exception = assertThrows(IllegalArgumentException.class, () -> {
        catalogoController.aggiungiVideogioco(videogioco, admin);
    });
    assertEquals("Videogioco gia' esistente", exception.getMessage());
}

@Test
void testCercaVideogiochi_Successo() {
    List<Videogioco> result = catalogoController.cercaVideogiochi("Platform", null,
        null);
    assertFalse(result.isEmpty());
    assertEquals("Super Mario Odyssey", result.get(0).getTitolo());
}

@Test
void testCercaVideogiochi_Fallimento_NessunRisultato() {
    List<Videogioco> result = catalogoController.cercaVideogiochi("NonEsistente",
        null, null);
    assertTrue(result.isEmpty());
}
```

Snippet 29: Test di CatalogoController

#### 4.1.3 CarrelloControllerTest

I test in `CarrelloControllerTest` verificano il comportamento delle operazioni legate alla gestione del carrello degli utenti, come l'aggiunta e la rimozione di videogiochi, l'esecuzione degli acquisti e la gestione degli observer per le notifiche, rispetto ai requisiti funzionali della piattaforma.

I test implementati in `CarrelloControllerTest` coprono i seguenti scenari:

- **testAggiungiAlCarrello\_Successo:** Verifica l'aggiunta di un videogioco (es. "Super Mario Odyssey") al carrello di un utente autenticato (es. "mario"). Si aspetta che il metodo `aggiungiAlCarrello` completi l'operazione senza errori e che il videogioco sia presente nel carrello.
- **testRimuoviDalCarrello\_Successo:** Verifica la rimozione di un videogioco (es. "Super Mario Odyssey") dal carrello di un utente (es. "mario"). Si aspetta che il metodo `rimuoviDalCarrello` completi l'operazione senza errori e che il videogioco non sia più presente nel carrello.

- **testEseguiAcquisto\_Successo:** Verifica l'esecuzione di un acquisto per un carrello di un utente con fondi sufficienti (es. "luigi" con fondo di 220.0). Si aspetta che il metodo `eseguiAcquisto` restituisca `true` e che il fondo dell'utente venga ridotto correttamente (es.  $220.00 - 69.99$ ).
- **testEseguiAcquisto\_Fallimento\_FondoInsufficiente:** Testa il tentativo di eseguire un acquisto con fondi insufficienti (es. "mario" con videogioco da 59.99). Si aspetta che venga lanciata un'`IllegalArgumentException`, garantendo che la piattaforma impedisca acquisti non validi.
- **testEseguiAcquisto\_Fallimento\_VideogiocoGiaInLibreria:** Testa il tentativo di acquistare un videogioco già presente nella libreria dell'utente (es. "luigi" tenta di riacquistare "Super Mario Odyssey"). Si aspetta che venga lanciata un'`IllegalArgumentException`, garantendo che la piattaforma prevenga acquisti duplicati.
- **testAggiungiRimuoviObserver\_Successo:** Verifica l'aggiunta e la rimozione di un observer (es. `LibreriaController`) al carrello. Si aspetta che i metodi `aggiungiObserver` e `rimuoviObserver` completino le operazioni senza errori.
- **testNotificaAcquistoCompletato\_Successo:** Verifica la notifica di un acquisto completato a un observer per un carrello di un utente (es. "mario"). Si aspetta che il metodo `notificaAcquistoCompletato` completi l'operazione senza errori.

```
@Test
void testAggiungiAlCarrello_Successo() {
    Utente utente = utenteController.autenticaUtente("mario", "password123");
    Videogioco videogioco = new Videogioco(1, "Super Mario Odyssey", "Platform", "
        Nintendo Switch", 59.99, LocalDate.parse("2017-10-27"), false);
    assertDoesNotThrow(() -> carrelloController.aggiungiAlCarrello(utente,
        videogioco));
    Carrello carrello = carrelloController.getCarrello(utente);
    assertTrue(carrello.getVideogiochi().stream().anyMatch(v -> v.getId() == 1));
}
```

```
@Test
void testRimuoviDalCarrello_Successo() {
    Utente utente = utenteController.autenticaUtente("mario", "password123");
    Videogioco videogioco = new Videogioco(1, "Super Mario Odyssey", "Platform", "
        Nintendo Switch", 59.99, LocalDate.parse("2017-10-27"), false);
    carrelloController.aggiungiAlCarrello(utente, videogioco);
    assertDoesNotThrow(() -> carrelloController.rimuoviDalCarrello(utente,
        videogioco));
    Carrello carrello = carrelloController.getCarrello(utente);
    assertFalse(carrello.getVideogiochi().stream().anyMatch(v -> v.getId() == 1));
}
```

```
@Test
void testEseguiAcquisto_Successo() {
    Utente utente = utenteController.autenticaUtente("luigi", "password456");
    utenteController.ricaricaFondo(utente, 200.0);
    Carrello carrello = carrelloController.getCarrello(utente);
    boolean result = carrelloController.eseguiAcquisto(carrello);
    assertTrue(result);
    Utente aggiornato = utenteController.autenticaUtente("luigi", "password456");
    assertEquals(220.00 - 69.99, aggiornato.getFondo(), 0.01);
}
```

```
@Test
void testEseguiAcquisto_Fallimento_FondoInsufficiente() {
    Utente utente = utenteController.autenticaUtente("mario", "password123");
    Videogioco videogioco = new Videogioco(1, "Super Mario Odyssey", "Platform", "
        Nintendo Switch", 59.99, LocalDate.parse("2017-10-27"), false);
    carrelloController.aggiungiAlCarrello(utente, videogioco);
    Carrello carrello = carrelloController.getCarrello(utente);
}
```

```

        assertThrows(IllegalArgumentException.class, () -> carrelloController.
            eseguiAcquisto(carrello));
    }

    @Test
    void testEseguiAcquisto_Fallimento_VideogiocoGiaInLibreria() {
        Utente utente = utenteController.autenticaUtente("luigi", "password456");
        utenteController.ricaricaFondo(utente, 200.0);
        Videogioco videogioco = new Videogioco(1, "Super Mario Odyssey", "Platform", "
            Nintendo Switch", 59.99, LocalDate.parse("2017-10-27"), false);
        carrelloController.aggiungiAlCarrello(utente, videogioco);
        Carrello carrello = carrelloController.getCarrello(utente);
        carrelloController.eseguiAcquisto(carrello);
        carrelloController.aggiungiAlCarrello(utente, videogioco);
        assertThrows(IllegalArgumentException.class, () -> carrelloController.
            eseguiAcquisto(carrello));
    }

    @Test
    void testAggiungiRimuoviObserver_Successo() {
        Observer observer = new LibreriaController(new Libreria(0, 1), new
            StatisticheController());
        assertDoesNotThrow(() -> carrelloController.aggiungiObserver(observer));
        assertDoesNotThrow(() -> carrelloController.rimuoviObserver(observer));
    }

    @Test
    void testNotificaAcquistoCompletato_Successo() {
        Utente utente = utenteController.autenticaUtente("mario", "password123");
        Carrello carrello = new Carrello(0, utente.getId());
        Observer observer = new LibreriaController(new Libreria(0, utente.getId()), new
            StatisticheController());
        carrelloController.aggiungiObserver(observer);
        assertDoesNotThrow(() -> carrelloController.notificaAcquistoCompletato(utente,
            carrello));
    }

```

Snippet 30: Test di CarrelloController

#### 4.1.4 LibreriaControllerTest

I test in `LibreriaControllerTest` verificano il comportamento delle operazioni legate alla gestione della libreria di videogiochi di un utente, come il caricamento della libreria, l'installazione e la disinstallazione dei videogiochi, e la gestione degli acquisti.

I test implementati in `LibreriaControllerTest` coprono i seguenti scenari:

- **testCaricaLibreria\_Successo:** Verifica il caricamento della libreria di un utente autenticato (es. "toad"). Si aspetta che il metodo `caricaLibreria` completi l'operazione senza errori e che l'ID dell'utente corrisponda a quello della libreria caricata.
- **testInstallaVideogioco\_Successo:** Verifica l'installazione di un videogioco acquistato (es. "Super Mario Odyssey") nella libreria di un utente (es. "luigi"). Si aspetta che il metodo `installaVideogioco` completi l'operazione senza errori e che il videogioco risulti installato.
- **testInstallaVideogioco\_Fallimento\_VideogiocoGiaInstallato:** Testa il tentativo di installare un videogioco già installato (es. "Super Mario Odyssey") nella libreria di un utente (es. "mario"). Si aspetta che venga lanciata un'`IllegalStateException`, garantendo che la piattaforma gestisca correttamente lo stato dei videogiochi.
- **testDisinstallaVideogioco\_Successo:** Verifica la disinstallazione di un videogioco installato (es. "Super Mario Odyssey") dalla libreria di un utente (es. "mario"). Si aspetta che il metodo `disinstallaVideogioco` completi l'operazione senza errori e che il videogioco non risulti più installato.

- **testDisinstallaVideogioco\_Fallimento\_VideogiocoNonInstallato:** Testa il tentativo di disinstallare un videogioco non installato (es. "Fortnite") dalla libreria di un utente (es. "mario"). Si aspetta che venga lanciata un'IllegalStateException, garantendo la gestione degli errori.
- **testOnAcquistoCompletato\_Successo:** Verifica l'aggiornamento della libreria di un utente (es. "toad") in seguito a un acquisto completato (es. "Super Mario Odyssey"). Si aspetta che il metodo eseguiAcquisto del CarrelloController notifichi correttamente la libreria e che il videogioco risulti acquistato.

```

@Test
void testCaricaLibreria_Successo() {
    Utente toad = utenteController.autenticaUtente("toad", "password101");
    assertDoesNotThrow(() -> libreriaController.caricaLibreria(toad));
    assertEquals(toad.getId(), libreriaController.getLibreria().getIdUtente());
}

@Test
void testInstallaVideogioco_Successo() {
    Utente luigi = utenteController.autenticaUtente("luigi", "password456");
    utenteController.ricaricaFondo(luigi, 200.0);
    Videomodosey = catalogoController.getVideogiochiPerUtente(null)
        .stream()
        .filter(v -> v.getId() == 1)
        .findFirst()
        .orElseThrow(() -> new IllegalStateException("Videogioco con ID 1 non trovato"));
    carrelloController.aggiungiAlCarrello(luigi, superMarioOdyssey);
    Carrello carrello = carrelloController.getCarrello(luigi);
    carrelloController.eseguiAcquisto(carrello);
    libreriaController.caricaLibreria(luigi);
    assertDoesNotThrow(() -> libreriaController.installaVideogioco(luigi, superMarioOdyssey));
    assertTrue(libreriaController.getLibreria().isVideogiocoInstallato(superMarioOdyssey));
}

@Test
void testInstallaVideogioco_Fallimento_VideogiocoGiaInstallato() {
    Utente mario = utenteController.autenticaUtente("mario", "password123");
    Videogioco superMarioOdyssey = catalogoController.getVideogiochiPerUtente(null)
        .stream()
        .filter(v -> v.getId() == 1)
        .findFirst()
        .orElseThrow(() -> new IllegalStateException("Videogioco con ID 1 non trovato"));
    libreriaController.caricaLibreria(mario);
    assertThrows(IllegalStateException.class, () -> libreriaController.installaVideogioco(mario, superMarioOdyssey));
}

@Test
void testDisinstallaVideogioco_Successo() {
    Utente mario = utenteController.autenticaUtente("mario", "password123");
    Videogioco superMarioOdyssey = catalogoController.getVideogiochiPerUtente(null)
        .stream()
        .filter(v -> v.getId() == 1)
        .findFirst()
        .orElseThrow(() -> new IllegalStateException("Videogioco con ID 1 non trovato"));
    libreriaController.caricaLibreria(mario);
    assertDoesNotThrow(() -> libreriaController.disinstallaVideogioco(mario, superMarioOdyssey));
}

```

```

        assertFalse(libreriaController.getLibreria().isVideogiocoInstallato(
            superMarioOdyssey));
    }

    @Test
    void testDisinstallaVideogioco_Fallimento_VideogiocoNonInstallato() {
        Utente mario = utenteController.autenticaUtente("mario", "password123");
        Videogioco fortnite = catalogoController.getVideogiochiPerUtente(null)
            .stream()
            .filter(v -> v.getId() == 3)
            .findFirst()
            .orElseThrow(() -> new IllegalStateException("Videogioco con ID 3 non
                trovato"));
        libreriaController.caricaLibreria(mario);
        assertThrows(IllegalStateException.class, () -> libreriaController.
            disinstallaVideogioco(mario, fortnite));
    }

    @Test
    void testOnAcquistoCompletato_Successo() {
        Utente toad = utenteController.autenticaUtente("toad", "password101");
        Videogioco superMarioOdyssey = catalogoController.getVideogiochiPerUtente(null)
            .stream()
            .filter(v -> v.getId() == 1)
            .findFirst()
            .orElseThrow(() -> new IllegalStateException("Videogioco con ID 1 non
                trovato"));
        utenteController.ricaricaFondo(toad, 100.0);
        libreriaController.caricaLibreria(toad);
        carrelloController.aggiungiAlCarrello(toad, superMarioOdyssey);
        Carrello carrello = carrelloController.getCarrello(toad);
        carrelloController.aggiungiObserver(libreriaController);
        assertDoesNotThrow(() -> carrelloController.eseguiAcquisto(carrello));
        assertTrue(libreriaController.getLibreria().getVideogiochiAcquistati().stream().
            anyMatch(v -> v.getId() == 1));
    }

```

Snippet 31: Test di LibreriaController

#### 4.1.5 StatisticheControllerTest

I test in `StatisticheControllerTest` verificano il comportamento delle operazioni legate alla gestione delle statistiche di gioco degli utenti, come l'aggiornamento del tempo di gioco e il recupero di statistiche e achievement.

I test implementati in `StatisticheControllerTest` coprono i seguenti scenari:

- **testGetStatistiche\_Successo:** Verifica il recupero delle statistiche di un videogioco per un utente (es. "mario", "Super Mario Odyssey"). Si aspetta che il metodo `getStatistiche` restituisca un oggetto `StatisticheVideogioco` non nullo con il tempo di gioco atteso (es. 7200 secondi).
- **testGetStatistiche\_Fallimento\_NessunaStatistica:** Testa il recupero delle statistiche per un videogioco non associato a un utente (es. "luigi", "Super Mario Odyssey"). Si aspetta che il metodo `getStatistiche` restituisca null, indicando l'assenza di statistiche.
- **testGetUtenteAchievements\_Successo:** Verifica il recupero degli achievement di un videogioco per un utente (es. "mario", "Super Mario Odyssey"). Si aspetta che il metodo `getUtenteAchievements` restituisca una lista non vuota contenente achievement come "Primo Boss".
- **testGetUtenteAchievements\_Fallimento\_NessunAchievement:** Testa il recupero degli achievement per un videogioco non associato a un utente (es. "luigi", "Super Mario Odyssey"). Si aspetta che il metodo `getUtenteAchievements` restituisca una lista vuota.

```

@Test
void testGetStatistiche_Successo() {
    Utente utente = new Utente(1, "mario", "password123", "mario@email.com", true,
        50.0);
    Videogioco videogioco = new Videogioco(1, "Super Mario Odyssey", "Platform", "
        Nintendo Switch", 59.99, LocalDate.parse("2017-10-27"), false);
    StatisticheVideogioco statistiche = statisticheController.getStatistiche(utente,
        videogioco);
    assertNotNull(statistiche);
    assertEquals(7200, statistiche.getTempoGioco());
}

@Test
void testGetStatistiche_Fallimento_NessunaStatistica() {
    Utente utente = new Utente(2, "luigi", "password456", "luigi@email.com", false,
        20.0);
    Videogioco videogioco = new Videogioco(1, "Super Mario Odyssey", "Platform", "
        Nintendo Switch", 59.99, LocalDate.parse("2017-10-27"), false);
    StatisticheVideogioco statistiche = statisticheController.getStatistiche(utente,
        videogioco);
    assertNull(statistiche);
}

@Test
void testGetUtenteAchievements_Successo() {
    Utente utente = new Utente(1, "mario", "password123", "mario@email.com", true,
        50.0);
    Videogioco videogioco = new Videogioco(1, "Super Mario Odyssey", "Platform", "
        Nintendo Switch", 59.99, LocalDate.parse("2017-10-27"), false);
    List<Achievement> achievements = statisticheController.getUtenteAchievements(
        utente, videogioco);
    assertFalse(achievements.isEmpty());
    assertTrue(achievements.stream().anyMatch(a -> a.getNome().equals("Primo Boss"))
    );
}

@Test
void testGetUtenteAchievements_Fallimento_NessunAchievement() {
    Utente utente = new Utente(2, "luigi", "password456", "luigi@email.com", false,
        20.0);
    Videogioco videogioco = new Videogioco(1, "Super Mario Odyssey", "Platform", "
        Nintendo Switch", 59.99, LocalDate.parse("2017-10-27"), false);
    List<Achievement> achievements = statisticheController.getUtenteAchievements(
        utente, videogioco);
    assertTrue(achievements.isEmpty());
}

```

Snippet 32: Test di StatisticheController

#### 4.1.6 AchievementControllerTest

I test in `AchievementControllerTest` verificano il comportamento delle operazioni legate alla gestione degli achievement dei videogiochi, come l'aggiunta, la modifica, l'eliminazione e il recupero degli achievement.

I test implementati in `AchievementControllerTest` coprono i seguenti scenari:

- **testAggiungiAchievement\_Successo:** Verifica l'aggiunta di un nuovo achievement (es. "Completato Livello 1") per un videogioco (es. ID 1) da parte di un amministratore autenticato (es. "admin"). Si aspetta che il metodo `aggiungiAchievement` completi l'operazione senza errori e che l'achievement sia recuperabile.
- **testAggiungiAchievement\_Fallimento\_AdminNonAutenticato:** Testa il tentativo di aggiungere un achievement da parte di un amministratore non autenticato (es. "fakeAdmin"). Si aspetta che venga lanciata un'`SecurityException`, garantendo che la piattaforma impedisca modifiche non autorizzate.

- **testModificaAchievement\_Successo:** Verifica la modifica di un achievement esistente (es. "Completato Livello 1") con nuovi dati (es. "Completato Livello 1 Aggiornato", tempo richiesto 1200). Si aspetta che il metodo `modificaAchievement` aggiorni correttamente l'achievement.
- **testGetAchievementsByVideogioco\_Fallimento\_NessunAchievement:** Testa il recupero degli achievement per un videogioco senza achievement associati (es. ID 999). Si aspetta che il metodo `getAchievementsByVideogioco` restituisca una lista vuota.

```
@Test
void testAggiungiAchievement_Successo() {
    Admin admin = adminController.autenticaAdmin("admin", "admin123");
    assertDoesNotThrow(() -> achievementController.aggiungiAchievement(1, "
        Completato Livello 1", "Completa il primo livello", 600, admin));
    List<Achievement> achievements = achievementController.
        getAchievementsByVideogioco(1);
    assertFalse(achievements.isEmpty());
    assertTrue(achievements.stream().anyMatch(a -> a.getNome().equals("Completato
        Livello 1"))));
}

@Test
void testAggiungiAchievement_Fallimento_AdminNonAutenticato() {
    Admin admin = new Admin(999, "fakeAdmin", "fakePass");
    assertThrows(SecurityException.class, () -> achievementController.
        aggiungiAchievement(1, "Completato Livello 1", "Completa il primo livello",
        600, admin));
}

@Test
void testGetAchievementsByVideogioco_Successo() {
    Admin admin = adminController.autenticaAdmin("admin", "admin123");
    achievementController.aggiungiAchievement(1, "Completato Livello 1", "Completa
        il primo livello", 600, admin);
    List<Achievement> achievements = achievementController.
        getAchievementsByVideogioco(1);
    assertFalse(achievements.isEmpty());
    assertTrue(achievements.stream().anyMatch(a -> a.getNome().equals("Completato
        Livello 1"))));
}

@Test
void testGetAchievementsByVideogioco_Fallimento_NessunAchievement() {
    List<Achievement> achievements = achievementController.
        getAchievementsByVideogioco(999);
    assertTrue(achievements.isEmpty());
}
```

Snippet 33: Test di AchievementController

#### 4.1.7 AdminControllerTest

I test in `AdminControllerTest` verificano il comportamento delle operazioni legate alla gestione degli amministratori, come la registrazione e l'autenticazione.

I test implementati in `AdminControllerTest` coprono i seguenti scenari:

- **testRegistraAdmin\_Fallimento\_UsernameEsistente:** Testa il tentativo di registrare un amministratore con un username già esistente (es. "admin"). Si aspetta che il metodo `registraAdmin` restituisca `false`, garantendo che la piattaforma impedisca registrazioni duplicate.
- **testAutenticaAdmin\_Successo:** Verifica l'autenticazione di un amministratore con credenziali corrette (es. "admin", "admin123"). Si aspetta che il metodo `autenticaAdmin` restituisca un oggetto `Admin` non nullo con l'username corretto.

- **testAutenticaAdmin\_Fallimento.UsernameNonEsistente:** Testa il tentativo di autenticazione di un amministratore con un username inesistente (es. "nonEsistente"). Si aspetta che il metodo `autenticaAdmin` restituisca `null`, garantendo la gestione degli errori di autenticazione.

```
@Test
void testRegistraAdmin_Fallimento_UsernameEsistente() {
    Admin admin = new Admin(0, "admin", "nuovaPass");
    boolean result = adminController.registraAdmin(admin);
    assertFalse(result);
}

@Test
void testAutenticaAdmin_Successo() {
    Admin admin = adminController.autenticaAdmin("admin", "admin123");
    assertNotNull(admin);
    assertEquals("admin", admin.getUsername());
}

@Test
void testAutenticaAdmin_Fallimento_UsernameNonEsistente() {
    Admin admin = adminController.autenticaAdmin("nonEsistente", "pass");
    assertNull(admin);
}
```

Snippet 34: Test di AdminController

## 4.2 Domain Model Test

Il modello di dominio della piattaforma è costituito da classi semplici che rappresentano le entità principali del sistema, come `Utente`, `Videogioco`, `Carrello`, `Abbonamento`, `Libreria`, `StatisticheVideogioco` e `Achievement`. Queste classi sono progettate principalmente come strutture dati, implementando metodi `getter` e `setter` per accedere e modificare gli attributi. Data la loro semplicità e il ruolo di contenitori di dati, si è scelto di limitare i test unitari a una singola classe del modello di dominio, la classe `Carrello`, che riveste un ruolo centrale nella gestione del calcolo del totale dei videogiochi acquistati. In particolare, i test su `Carrello` verificano il corretto funzionamento del pattern *Strategy* utilizzato per determinare i prezzi in base al tipo di abbonamento dell'utente (es. `NonAbbonato`, `Silver`, `Gold`), garantendo che il calcolo del totale rifletta correttamente gli sconti applicabili e la gestione di videogiochi gratuiti o a pagamento.

### 4.2.1 CarrelloTest

I test in `CarrelloTest` verificano il comportamento della classe `Carrello`, con particolare attenzione alle operazioni di aggiunta e rimozione di videogiochi e al calcolo del totale del carrello in base al pattern *Strategy*.

I test implementati in `CarrelloTest` coprono i seguenti scenari:

- **testAddVideogioco\_Successo\_NuovoVideogioco:** Verifica l'aggiunta di un nuovo videogioco (es. "Super Mario Odyssey") al carrello. Si aspetta che il metodo `addVideogioco` aggiunga correttamente il videogioco e che la lista dei videogiochi contenga un solo elemento.
- **testAddVideogioco\_Fallimento\_VideogiocoDuplicato:** Testa il tentativo di aggiungere un videogioco già presente nel carrello (es. "Super Mario Odyssey"). Si aspetta che il metodo `addVideogioco` non aggiunga duplicati, mantenendo la lista dei videogiochi con un solo elemento.
- **testRemoveVideogioco\_Successo:** Verifica la rimozione di un videogioco presente nel carrello (es. "Super Mario Odyssey"). Si aspetta che il metodo `removeVideogioco` rimuova correttamente il videogioco, lasciando il carrello vuoto.
- **testSetPrezzoStrategy\_Successo:** Verifica l'impostazione della strategia di prezzo `GoldAbbonatoPrezzoStrategy` e il calcolo del totale per un videogioco a pagamento (es. "Super Mario Odyssey"). Si aspetta che il metodo `getTotale` restituisca il prezzo scontato del 20% (es.  $59.99 * 0.8$ ).

- **testGetTotale\_NonAbbonato\_GiocoAPagamento:** Verifica il calcolo del totale per un carrello contenente un videogioco a pagamento (es. "Super Mario Odyssey") con strategia NonAbbonatoPrezzoStrategy. Si aspetta che il metodo `getTotale` restituisca il prezzo pieno (es. 59.99).
- **testGetTotale\_GoldAbbonato\_GiocoAPagamento:** Verifica il calcolo del totale per un carrello contenente un videogioco a pagamento (es. "Super Mario Odyssey") con strategia GoldAbbonatoPrezzoStrategy. Si aspetta che il metodo `getTotale` restituisca il prezzo scontato del 20% (es.  $59.99 * 0.8$ ).
- **testGetTotale\_GoldAbbonato\_GiocoGratuito:** Verifica il calcolo del totale per un carrello contenente un videogioco gratuito (es. "Fortnite"). Si aspetta che il metodo `getTotale` restituisca 0.0.

```

@Test
void testAddVideogioco_Successo_NuovoVideogioco() {
    carrello.addVideogioco(marioOdyssey);
    List<Videogioco> videogiochi = carrello.getVideogiochi();
    assertEquals(1, videogiochi.size());
    assertTrue(videogiochi.contains(marioOdyssey));
}

@Test
void testAddVideogioco_Fallimento_VideogiocoDuplicato() {
    carrello.addVideogioco(marioOdyssey);
    carrello.addVideogioco(marioOdyssey);
    List<Videogioco> videogiochi = carrello.getVideogiochi();
    assertEquals(1, videogiochi.size());
    assertTrue(videogiochi.contains(marioOdyssey));
}

@Test
void testRemoveVideogioco_Successo() {
    carrello.addVideogioco(marioOdyssey);
    carrello.removeVideogioco(marioOdyssey);
    List<Videogioco> videogiochi = carrello.getVideogiochi();
    assertTrue(videogiochi.isEmpty());
}

@Test
void testSetPrezzoStrategy_Successo() {
    PrezzoStrategy goldStrategy = new GoldAbbonatoPrezzoStrategy();
    carrello.setPrezzoStrategy(goldStrategy);
    carrello.addVideogioco(marioOdyssey);
    double totale = carrello.getTotale(utente);
    assertEquals(59.99 * (1 - 0.2), totale, 0.01);
}

@Test
void testGetTotale_NonAbbonato_GiocoAPagamento() {
    carrello.setPrezzoStrategy(new NonAbbonatoPrezzoStrategy());
    carrello.addVideogioco(marioOdyssey);
    double totale = carrello.getTotale(utente);
    assertEquals(59.99, totale, 0.01);
}

@Test
void testGetTotale_GoldAbbonato_GiocoAPagamento() {
    carrello.setPrezzoStrategy(new GoldAbbonatoPrezzoStrategy());
    carrello.addVideogioco(marioOdyssey);
    double totale = carrello.getTotale(utente);
    assertEquals(59.99 * (1 - 0.2), totale, 0.01);
}

```

```

@Test
void testGetTotale_GoldAbbonato_GiocoGratuito() {
    carrello.setPrezzoStrategy(new GoldAbbonatoPrezzoStrategy());
    carrello.addVideogioco(fortnite);
    double totale = carrello.getTotale(utente);
    assertEquals(0.0, totale, 0.01);
}

```

Snippet 35: Test di Carrello

### 4.3 Test del Pacchetto ORM

I test delle classi DAO verificano il corretto funzionamento delle operazioni di creazione, lettura, aggiornamento ed eliminazione (CRUD) per le entità del Domain Model, garantendo che i dati siano correttamente salvati, recuperati, aggiornati ed eliminati nel database. Per assicurare la coerenza dei dati durante i test, ogni metodo di test è preceduto e seguito dall'esecuzione dei file SQL `reset.sql`, `ProgettoSWE.sql` e `InserimentoProgettoSWE.sql`, come descritto nella sezione dedicata al database. I test si concentrano principalmente sui metodi che interagiscono direttamente con il database, come `save`, `update`, `findById`, `findByCriteria` e `delete`, poiché questi rappresentano i punti critici per la persistenza e il recupero dei dati. Particolare attenzione è stata posta ai casi di fallimento (es. recupero di entità inesistenti o violazioni di vincoli) per verificare la robustezza della piattaforma nel gestire errori di interazione con il database.

#### 4.3.1 VideogiocoDAOTest

I test in `VideogiocoDAOTest` verificano il comportamento delle operazioni CRUD per l'entità `Videogioco`, come il salvataggio, l'aggiornamento, il recupero e l'eliminazione di videogiochi nel database.

I test implementati in `VideogiocoDAOTest` coprono i seguenti scenari:

- **testSave\_Successo:** Verifica il salvataggio di un nuovo videogioco (es. "Animal Crossing") nel database. Si aspetta che il metodo `save` restituisca un oggetto `Videogioco` non nullo con un ID valido e attributi corretti (es. prezzo 59.99).
- **testUpdate\_Successo:** Verifica l'aggiornamento di un videogioco esistente (es. ID 1, "Super Mario Odyssey") con un nuovo prezzo (es. 49.99). Si aspetta che il metodo `update` completi l'operazione senza errori e che il videogioco aggiornato rifletta il nuovo prezzo.
- **testFindById\_Successo:** Verifica il recupero di un videogioco per ID (es. ID 1, "Super Mario Odyssey"). Si aspetta che il metodo `findById` restituisca un oggetto `Videogioco` non nullo con gli attributi attesi.
- **testFindById\_Fallimento\_IdNonEsistente:** Testa il recupero di un videogioco con un ID inesistente (es. ID 999). Si aspetta che il metodo `findById` restituisca `null`, garantendo la gestione degli errori.
- **testFindByCriteria\_Successo\_TuttiCriteri:** Verifica la ricerca di videogiochi utilizzando tutti i criteri (es. genere "Platform", piattaforma "Nintendo Switch", data 2017-10-27). Si aspetta che il metodo `findByCriteria` restituisca una lista non vuota contenente videogiochi come "Super Mario Odyssey".
- **testFindByCriteria\_Fallimento\_NessunRisultato:** Testa la ricerca di videogiochi con un genere inesistente (es. "NonEsistente"). Si aspetta che il metodo `findByCriteria` restituisca una lista vuota, indicando l'assenza di risultati.
- **testDelete\_Successo:** Verifica l'eliminazione di un videogioco dal database (es. "Test Game"). Si aspetta che il metodo `delete` completi l'operazione senza errori e che il videogioco non sia più recuperabile tramite `findById`.

```

@Test
void testSave_Successo() {
    Videogioco videogioco = new Videogioco(0, "Animal Crossing", "Simulazione", "
        Nintendo Switch", 59.99, LocalDate.parse("2020-03-20"), false);
    Videogioco saved = videogiocoDAO.save(videogioco);
}

```

```

        assertNotNull(saved);
        assertTrue(saved.getId() > 0);
        assertEquals("Animal Crossing", saved.getTitolo());
        assertEquals(59.99, saved.getPrezzo(), 0.01);
    }

    @Test
    void testUpdate_Successo() {
        Videogioco videogioco = videogiocoDAO.findById(1);
        videogioco.setPrezzo(49.99);
        assertDoesNotThrow(() -> videogiocoDAO.update(videogioco));
        Videogioco updated = videogiocoDAO.findById(1);
        assertEquals(49.99, updated.getPrezzo(), 0.01);
    }

    @Test
    void testFindById_Successo() {
        Videogioco videogioco = videogiocoDAO.findById(1);
        assertNotNull(videogioco);
        assertEquals("Super Mario Odyssey", videogioco.getTitolo());
        assertEquals(59.99, videogioco.getPrezzo(), 0.01);
    }

    @Test
    void testFindById_Fallimento_IdNonEsistente() {
        Videogioco videogioco = videogiocoDAO.findById(999);
        assertNull(videogioco);
    }

    @Test
    void testFindByCriteria_Successo_TuttiCriteri() {
        List<Videogioco> videogiochi = videogiocoDAO.findByCriteria("Platform", "
            Nintendo Switch", LocalDate.parse("2017-10-27"));
        assertFalse(videogiochi.isEmpty());
        assertTrue(videogiochi.stream().anyMatch(v -> v.getTitolo().equals("Super Mario
            Odyssey")));
    }

    @Test
    void testFindByCriteria_Fallimento_NessunRisultato() {
        List<Videogioco> videogiochi = videogiocoDAO.findByCriteria("NonEsistente", null
            , null);
        assertTrue(videogiochi.isEmpty());
    }

    @Test
    void testDelete_Successo() {
        Videogioco videogioco = videogiocoDAO.save(new Videogioco(0, "Test Game", "Test "
            , "Test", 10.0, LocalDate.now(), false));
        assertDoesNotThrow(() -> videogiocoDAO.delete(videogioco.getId()));
        Videogioco deleted = videogiocoDAO.findById(videogioco.getId());
        assertNull(deleted);
    }

```

Snippet 36: Test di VideogiocoDAO

### 4.3.2 CarrelloDAOTest

I test in CarrelloDAOTest verificano il comportamento delle operazioni CRUD per l'entità Carrello, come il salvataggio, l'aggiornamento, il recupero e l'eliminazione di carrelli nel database.

I test implementati in CarrelloDAOTest coprono i seguenti scenari:

- **testSave\_Successo\_CarrelloEsistente:** Verifica il salvataggio di un carrello esistente (es. ID 1, utente "mario") con un nuovo videogioco (es. "Minecraft"). Si aspetta che il metodo `save` aggiorni correttamente il carrello, includendo il nuovo videogioco e rimuovendo quelli non più presenti.
- **testUpdate\_Successo:** Verifica l'aggiornamento di un carrello esistente (es. ID 1, utente "mario") con un nuovo videogioco (es. "Minecraft"). Si aspetta che il metodo `update` completi l'operazione senza errori e che il carrello aggiornato contenga il nuovo videogioco.
- **testFindByUtenteId\_Successo:** Verifica il recupero di un carrello per l'ID di un utente (es. ID 1, "mario"). Si aspetta che il metodo `findByUtenteId` restituisca un oggetto `Carrello` non nullo contenente videogiochi attesi (es. ID 2).
- **testFindByUtenteId\_Fallimento\_UtenteNonEsistente:** Testa il recupero di un carrello per un utente inesistente (es. ID 999). Si aspetta che il metodo `findByUtenteId` restituisca `null`, garantendo la gestione degli errori.
- **testDelete\_Successo:** Verifica l'eliminazione di un carrello dal database (es. ID 1, utente "mario"). Si aspetta che il metodo `delete` completi l'operazione senza errori e che il carrello non sia più recuperabile tramite `findByUtenteId`.

**Comportamento in caso di fallimento:** Il test `testFindByUtenteId_Fallimento_UtenteNonEsistente` verifica la robustezza della piattaforma nel gestire il recupero di carrelli per utenti inesistenti, restituendo `null` in modo appropriato. Questo controllo assicura che il `CarrelloDAO` gestisca correttamente l'assenza di dati senza generare errori.

```
@Test
void testSave_Successo_CarrelloEsistente() {
    Carrello carrello = new Carrello(1, 1);
    Videogioco minecraft = videogiocoDAO.findById(5);
    carrello.addVideogioco(minecraft);
    assertDoesNotThrow(() -> carrelloDAO.save(carrello));
    Carrello saved = carrelloDAO.findByUtenteId(1);
    assertEquals(1, saved.getIdUtente());
    assertTrue(saved.getVideogiochi().stream().anyMatch(v -> v.getId() == 5));
    assertFalse(saved.getVideogiochi().stream().anyMatch(v -> v.getId() == 2));
}

@Test
void testUpdate_Successo() {
    Carrello carrello = carrelloDAO.findByUtenteId(1);
    carrello.addVideogioco(videogiocoDAO.findById(5));
    assertDoesNotThrow(() -> carrelloDAO.update(carrello));
    Carrello updated = carrelloDAO.findByUtenteId(1);
    assertTrue(updated.getVideogiochi().stream().anyMatch(v -> v.getId() == 5));
}

@Test
void testFindByUtenteId_Successo() {
    Carrello carrello = carrelloDAO.findByUtenteId(1);
    assertNotNull(carrello);
    assertEquals(1, carrello.getIdUtente());
    assertTrue(carrello.getVideogiochi().stream().anyMatch(v -> v.getId() == 2));
}

@Test
void testFindByUtenteId_Fallimento_UtenteNonEsistente() {
    Carrello carrello = carrelloDAO.findByUtenteId(999);
    assertNull(carrello);
}

@Test
void testDelete_Successo() {
    Carrello carrello = carrelloDAO.findByUtenteId(1);
```

```

    assertDoesNotThrow(() -> carrelloDAO.delete(carrello.getId()));
    Carrello deleted = carrelloDAO.findByUtenteId(1);
    assertNull(deleted);
}

```

#### Snippet 37: Test di CarrelloDAO

### 4.3.3 LibreriaDAOTest

I test in `LibreriaDAOTest` verificano il comportamento delle operazioni CRUD per l'entità `Libreria`, come il salvataggio, il recupero e l'aggiornamento delle librerie degli utenti nel database.

I test implementati in `LibreriaDAOTest` coprono i seguenti scenari:

- **testSave\_Successo\_NuovaLibreria:** Verifica il salvataggio di una nuova libreria per un utente (es. "luigi", ID 2) contenente un videogioco (es. "FIFA 23") con stato di installazione. Si aspetta che il metodo `save` completi l'operazione senza errori e che la libreria recuperata contenga il videogioco con lo stato corretto.
- **testFindByUtenteId\_Successo:** Verifica il recupero di una libreria per l'ID di un utente (es. ID 1, "mario"). Si aspetta che il metodo `findByUtenteId` restituisca un oggetto `Libreria` non nullo contenente videogiochi attesi (es. "Super Mario Odyssey") con stato di installazione corretto.
- **testFindByUtenteId\_Fallimento\_UtenteNonEsistente:** Testa il recupero di una libreria per un utente inesistente (es. ID 999). Si aspetta che il metodo `findByUtenteId` restituisca `null`, garantendo la gestione degli errori.
- **testUpdate\_Successo:** Verifica l'aggiornamento di una libreria esistente (es. ID 1, utente "mario") con un nuovo videogioco installato (es. "Fortnite"). Si aspetta che il metodo `update` completi l'operazione senza errori e che la libreria aggiornata rifletta lo stato di installazione corretto.

```

@Test
void testSave_Successo_NuovaLibreria() {
    Libreria libreria = new Libreria(0, 2); // Luigi
    Videogioco fifa = videogiocoDAO.findById(4); // FIFA 23
    libreria.addVideogioco(fifa);
    libreria.setVideogiocoInstallato(fifa, true);
    assertDoesNotThrow(() -> libreriaDAO.save(libreria));
    Libreria saved = libreriaDAO.findByUtenteId(2);
    assertNotNull(saved);
    assertEquals(2, saved.getIdUtente());
    assertTrue(saved.getVideogiochiAcquistati().stream().anyMatch(v -> v.getId() ==
        4));
    assertTrue(saved.isVideogiocoInstallato(fifa));
}

@Test
void testFindByUtenteId_Successo() {
    Libreria libreria = libreriaDAO.findByUtenteId(1); // Mario
    assertNotNull(libreria);
    assertEquals(1, libreria.getIdUtente());
    assertTrue(libreria.getVideogiochiAcquistati().stream().anyMatch(v -> v.getId()
        == 1));
    assertTrue(libreria.isVideogiocoInstallato(videogiocoDAO.findById(1)));
}

@Test
void testFindByUtenteId_Fallimento_UtenteNonEsistente() {
    Libreria libreria = libreriaDAO.findByUtenteId(999);
    assertNull(libreria);
}

@Test
void testUpdate_Successo() {

```

```
Libreria libreria = libreriaDAO.findByUtenteId(1); // Mario
Videogioco fortnite = videogiocoDAO.findById(3); // Fortnite
libreria.setVideogiocoInstallato(fortnite, true);
assertDoesNotThrow(() -> libreriaDAO.update(libreria));
Libreria updated = libreriaDAO.findByUtenteId(1);
assertTrue(updated.isVideogiocoInstallato(fortnite));
}
```

Snippet 38: Test di LibreriaDAO