



UNIVERSITÀ DEGLI STUDI DI FIRENZE  
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

---

## SplitManager

Sistema software per la gestione delle spese condivise

---

### Autori

Carmen Possidente  
Matteo Gerlotti  
Roberta Donato

### Matricole

7115970  
7025024  
7113502

**Corso:** Ingegneria del Software  
**Docente:** Prof. Enrico Vicario

Anno Accademico 2025–2026

# Indice

<b>1</b>	<b>Introduzione</b>	<b>3</b>
1.1	Statement . . . . .	3
1.2	Architettura . . . . .	4
<b>2</b>	<b>Analisi dei Requisiti</b>	<b>5</b>
2.1	Use Case Diagram . . . . .	5
2.2	Use Case Templates . . . . .	7
2.3	Mockups . . . . .	15
2.4	Page Navigation Diagram . . . . .	17
<b>3</b>	<b>Progettazione</b>	<b>18</b>
3.1	Package Diagram . . . . .	18
3.2	Class Diagrams . . . . .	20
3.2.1	Domain Model . . . . .	20
3.2.2	ORM . . . . .	22
3.2.3	Service Layer . . . . .	24
3.2.4	Controller Layer . . . . .	26
3.3	Database . . . . .	27
3.3.1	Modello ER . . . . .	27
3.3.2	Schema Relazionale . . . . .	28
<b>4</b>	<b>Implementazione</b>	<b>29</b>
<b>5</b>	<b>Test</b>	<b>30</b>
5.1	Test Strutturali (White-Box) . . . . .	30
5.2	Test di Integrazione (Grey-Box) . . . . .	31
5.3	Test Funzionali (Black-Box) . . . . .	31
5.3.1	Scenario di Test . . . . .	31
5.3.2	Aspetti Architetturali . . . . .	32

## Elenco delle figure

1	Panoramica dell'architettura logica a livelli del sistema. . . . .	4
2	Use Case Diagram - User . . . . .	5
3	Use Case Diagram - Member . . . . .	5
4	Use Case Diagram - Admin . . . . .	6
5	Mockup #1 – User Dashboard . . . . .	15
6	Mockup #2 – Group Details Page . . . . .	15
7	Mockup #3 – Add Expense Modal . . . . .	16

8	Mockup #4 – Balances & Settlement . . . . .	16
9	Page Navigation Diagram . . . . .	17
10	Suddivisione del Domain Model in sotto-package . . . . .	18
11	Dipendenze tra i package Controller, Service, ORM e Util . . . . .	19
12	Exceptions Package . . . . .	20
13	Domain Model Class Diagram . . . . .	22
14	ORM Class Diagram . . . . .	23
15	Service Layer Class Diagram . . . . .	24
16	Controller Layer Class Diagram . . . . .	26
17	Modello ER . . . . .	27
18	Snippet 1 - BalanceTest . . . . .	30
19	Snippet 2 - ExpenseTest . . . . .	31
20	Snippet 3 - MembershipTest . . . . .	31
21	Snippet 4 - StubNavigator . . . . .	33

# 1 Introduzione

## 1.1 Statement

**SplitManager** è un sistema software per la gestione delle spese condivise all'interno di gruppi di persone. L'applicazione consente a gruppi di amici, colleghi, parenti di tenere in modo organizzato il conto delle spese comuni e di semplificare i rimborsi.

Ruoli e funzionalità principali:

- **User** (Utente Registrato): può registrarsi al sistema, effettuare il login, creare nuovi gruppi o unirsi a gruppi esistenti.
- **Member** (Membro del Gruppo): può inserire nuove spese specificando chi ha pagato e chi ne ha beneficiato, visualizzare la cronologia delle spese e i saldi del gruppo.
- **Admin** (Amministratore di Gruppo): può approvare nuovi membri, modificare/cancellare qualsiasi spesa del gruppo e gestire le impostazioni del gruppo.

I saldi sono per gran parte gestiti dal sistema: esso calcola in tempo reale il saldo netto di ogni utente e offre una rappresentazione chiara di chi deve e di chi vanta crediti all'interno del gruppo. Il calcolo avviene ottimizzando i rimborsi, in modo da minimizzare il numero di transazioni necessarie per chiudere tutti i conti.

## 1.2 Architettura

Il sistema è stato progettato seguendo un'architettura **multilayer**, che garantisce un netto disaccoppiamento tra l'interfaccia utente, la logica di business e la persistenza dei dati.

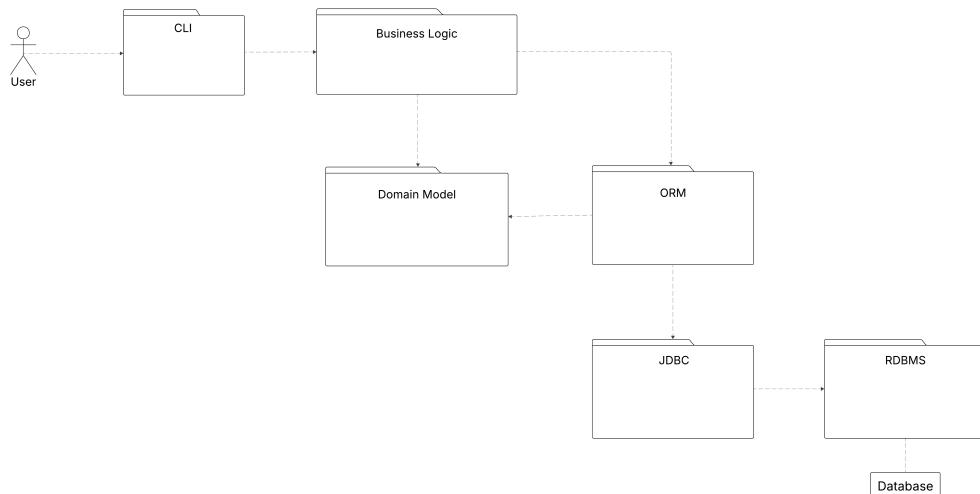


Figura 1: Panoramica dell'architettura logica a livelli del sistema.

## 2 Analisi dei Requisiti

Questa sezione descrive i requisiti funzionali del sistema attraverso modelli UML e artefatti di supporto alla progettazione. Tutti i diagrammi presenti in questa sezione e nella successiva sono stati realizzati con **Lucidchart**, mentre i mockup sono stati realizzati con il supporto dell'intelligenza artificiale generativa.

### 2.1 Use Case Diagram

I seguenti diagrammi rappresentano i tre attori del sistema (User, Member, Admin) e le principali funzionalità offerte dall'applicazione per ciascun ruolo.

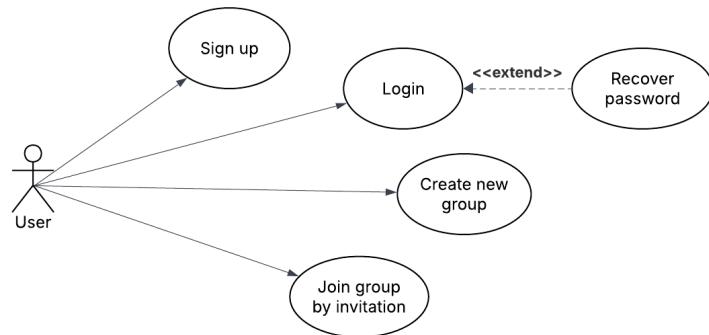


Figura 2: Use Case Diagram - User

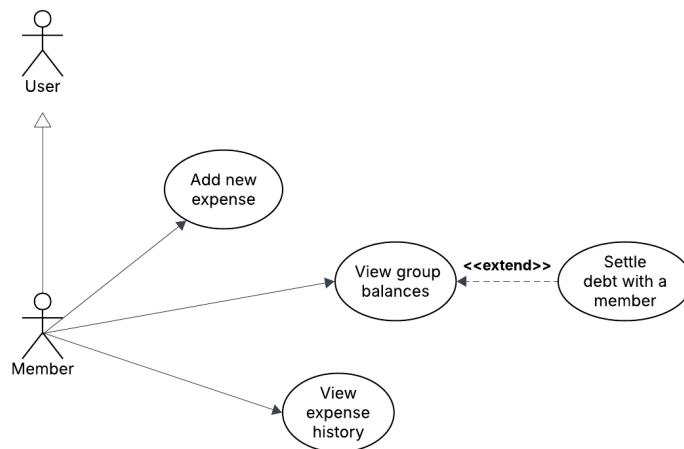


Figura 3: Use Case Diagram - Member

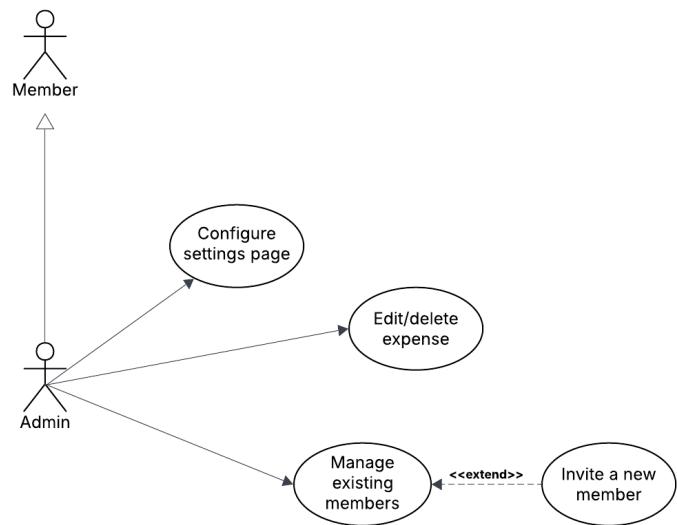


Figura 4: Use Case Diagram - Admin

## 2.2 Use Case Templates

Per ciascun caso d'uso identificato nel diagramma precedente, sono stati definiti i relativi template descrittivi. Ogni template specifica attore, pre-condizioni, flusso principale e flussi alternativi, fornendo una descrizione strutturata del comportamento atteso del sistema.

UC1 – Sign Up	
<b>Attore</b>	User
<b>Livello</b>	Function
<b>Pre-condizioni</b>	L'utente non deve essere già registrato nel sistema.
<b>Basic Flow</b>	<ol style="list-style-type: none"><li>1. L'utente accede alla pagina iniziale di SplitManager.</li><li>2. Clicca su "Sign up".</li><li>3. Il sistema apre la pagina "Create Account".</li><li>4. L'utente inserisce i dati richiesti:<ul style="list-style-type: none"><li>– Full name</li><li>– Email address</li><li>– Password</li><li>– Confirm password</li></ul></li><li>5. Il sistema verifica la validità dei dati (formato email, corrispondenza password).</li><li>6. Il sistema registra il nuovo utente nel database.</li><li>7. Il sistema mostra un messaggio di conferma e reindirizza alla pagina di login (Test #1).</li></ol>
<b>Alternative Flow</b>	<ol style="list-style-type: none"><li>5a. Se l'email è già registrata, il sistema mostra un messaggio di errore "Account already exists".</li><li>5b. Se i campi obbligatori non sono compilati o le password non coincidono, il sistema richiede la correzione.</li></ol>

Tabella 1: UC1 – Sign Up

UC2 – Login	
<b>Attore</b>	User
<b>Livello</b>	Function
<b>Pre-condizioni</b>	L’utente deve essere già registrato nel sistema.
<b>Basic Flow</b>	<ol style="list-style-type: none"> <li>1. L’utente accede alla pagina “Login”.</li> <li>2. Inserisce le proprie credenziali (email e password).</li> <li>3. Clicca su “Login”.</li> <li>4. Il sistema verifica le credenziali nel database.</li> <li>5. Se valide, il sistema apre la Home Page utente, dove può creare o unirsi a gruppi.</li> </ol>
<b>Alternative Flow</b>	<ol style="list-style-type: none"> <li>4a. Se l'email o la password non sono corrette, il sistema mostra un messaggio “Invalid credentials” e permette un nuovo tentativo.</li> <li>4b. Se l’utente dimentica la password, può cliccare su “Forgot password?” per avviare la procedura di recupero.</li> </ol>

Tabella 2: UC2 – Login

UC3 – Create New Group	
<b>Attore</b>	User
<b>Livello</b>	User Goal
<b>Pre-condizioni</b>	L’utente deve essere autenticato (logged in).
<b>Basic Flow</b>	<ol style="list-style-type: none"> <li>1. L’utente accede alla dashboard personale (<b>vedi Mockup #1</b>).</li> <li>2. Clicca su “Create New Group”.</li> <li>3. Il sistema apre la pagina di configurazione gruppo.</li> <li>4. L’utente inserisce i dettagli del gruppo: <ul style="list-style-type: none"> <li>– Group name</li> <li>– Description (facoltativa)</li> <li>– Currency (selezionabile da dropdown)</li> </ul> </li> <li>5. Clicca su “Create”.</li> <li>6. Il sistema registra il nuovo gruppo, imposta l’utente come Admin, e apre la pagina del gruppo creato.</li> </ol>
<b>Alternative Flow</b>	<ol style="list-style-type: none"> <li>5a. Se mancano dati obbligatori (es. nome gruppo), il sistema mostra un messaggio di errore “Please complete all required fields”.</li> <li>5b. Se si verifica un errore di rete o salvataggio, viene mostrato “Group creation failed. Try again later.”</li> </ol>

Tabella 3: UC3 – Create New Group

UC4 – Join Group by Invitation	
<b>Attore</b>	User
<b>Livello</b>	User Goal
<b>Pre-condizioni</b>	L'utente deve avere un account valido ed essere loggato. Deve aver ricevuto un link o un codice invito generato da un Admin.
<b>Basic Flow</b>	<ol style="list-style-type: none"> <li>1. L'utente accede alla sezione “Join Group” dalla Dashboard (<b>vedi Mockup #1</b>).</li> <li>2. Inserisce il codice di invito oppure clicca direttamente sul link ricevuto.</li> <li>3. Il sistema verifica la validità del codice e l'esistenza del gruppo.</li> <li>4. Se valido, il sistema aggiunge l'utente come Member del gruppo.</li> <li>5. Il sistema mostra un messaggio di conferma e reindirizza alla pagina del gruppo (<b>vedi Mockup #2</b>).</li> </ol>
<b>Alternative Flow</b>	<ol style="list-style-type: none"> <li>3a. Se il codice è errato o scaduto, il sistema mostra “Invalid or expired invitation”.</li> <li>3b. Se l'utente è già membro del gruppo, il sistema mostra “You are already part of this group”.</li> </ol>

Tabella 4: UC4 – Join Group by Invitation

UC5 – Add New Expense	
<b>Attore</b>	Member
<b>Livello</b>	User Goal
<b>Pre-condizioni</b>	Il membro deve appartenere ad almeno un gruppo.
<b>Basic Flow</b>	<ol style="list-style-type: none"> <li>Il membro accede alla pagina del gruppo.</li> <li>Clicca sul pulsante “Add Expense” in alto a destra.</li> <li>Il sistema apre la finestra modale di inserimento (<b>vedi Mockup #3</b>).</li> <li>Il membro inserisce i dettagli della spesa: <ul style="list-style-type: none"> <li>– Description</li> <li>– Amount</li> <li>– Category (tramite dropdown)</li> <li>– Paid by (tramite dropdown)</li> <li>– Split between (checkbox membri o “Select all”)</li> </ul> </li> <li>Il sistema verifica la correttezza dei dati inseriti.</li> <li>Il sistema registra la nuova spesa, aggiorna i saldi e chiude il modale tornando alla pagina del gruppo.</li> </ol>
<b>Alternative Flow</b>	<ol style="list-style-type: none"> <li>Se l'importo non è valido o i campi obbligatori sono vuoti, il sistema mostra un messaggio di errore.</li> <li>Se i dati sono incompleti, il sistema richiede il completamento dei campi mancanti.</li> </ol>

Tabella 5: UC5 – Add New Expense

UC6 – View Group Balances	
<b>Attore</b>	Member
<b>Livello</b>	User Goal
<b>Pre-condizioni</b>	Il membro deve appartenere ad almeno un gruppo.
<b>Basic Flow</b>	<ol style="list-style-type: none"> <li>Il membro accede alla pagina del gruppo.</li> <li>Selezione la sezione “Balances” (<b>vedi Mockup #4</b>).</li> <li>Il sistema mostra: <ul style="list-style-type: none"> <li>– Totale delle spese del gruppo</li> <li>– Saldo individuale di ciascun membro</li> <li>– Lista di debiti/crediti reciproci (“chi deve a chi”)</li> </ul> </li> <li>Il membro può visualizzare i dettagli delle transazioni passate.</li> </ol>
<b>Alternative Flow</b>	<ol style="list-style-type: none"> <li>Se non ci sono spese registrate, il sistema mostra “No expenses yet”.</li> </ol>

Tabella 6: UC6 – View Group Balances

UC7 – View Expense History	
<b>Attore</b>	Member
<b>Livello</b>	User Goal
<b>Pre-condizioni</b>	Il membro appartiene ad un gruppo con almeno una spesa registrata.
<b>Basic Flow</b>	<ol style="list-style-type: none"> <li>1. Il membro accede alla sezione “Expense History” del gruppo.</li> <li>2. Il sistema mostra una lista cronologica delle spese con: <ul style="list-style-type: none"> <li>– Data</li> <li>– Descrizione</li> <li>– Importo</li> <li>– Chi ha pagato</li> <li>– Tra chi è stata divisa</li> </ul> </li> <li>3. Il membro può filtrare o cercare spese specifiche per categoria, data o membro.</li> </ol>
<b>Alternative Flow</b>	<ol style="list-style-type: none"> <li>2a. Se non ci sono spese nel periodo selezionato, il sistema mostra “No expenses found”.</li> </ol>

Tabella 7: UC7 – View Expense History

UC8 – Settle Debt with a Member	
<b>Attore</b>	Member
<b>Livello</b>	User Goal
<b>Pre-condizioni</b>	Il membro ha un debito aperto verso un altro membro del gruppo.
<b>Basic Flow</b>	<ol style="list-style-type: none"> <li>1. Il membro accede alla sezione “Balances” (<b>vedi Mockup #4</b>).</li> <li>2. Seleziona il debito da saldare.</li> <li>3. Il sistema mostra l’importo dovuto e chiede conferma del pagamento.</li> <li>4. Il membro conferma.</li> <li>5. Il sistema registra il pagamento e aggiorna immediatamente i saldi mostrati nella pagina.</li> </ol>
<b>Alternative Flow</b>	<ol style="list-style-type: none"> <li>4a. Se il membro annulla l’operazione, il sistema chiude la richiesta senza registrare il pagamento.</li> </ol>

Tabella 8: UC8 – Settle Debt with a Member

UC9 – Invite a New Member	
<b>Attore</b>	Admin
<b>Livello</b>	User Goal
<b>Pre-condizioni</b>	L'Admin è autenticato e si trova nella pagina del gruppo.
<b>Basic Flow</b>	<ol style="list-style-type: none"> <li>1. L'Admin accede alla sezione "Members Page" del gruppo.</li> <li>2. Seleziona l'opzione "Invite a new member".</li> <li>3. Il sistema genera un codice/link di invito univoco.</li> <li>4. L'Admin copia o invia il link ai potenziali membri tramite canali esterni.</li> <li>5. Il sistema registra l'invito in stato "Waiting for acceptance".</li> <li>6. Quando l'utente destinatario utilizza il link, il sistema valida il codice e invia la richiesta di ingresso al gruppo.</li> <li>7. L'Admin riceve la notifica di nuova richiesta e potrà successivamente approvarla (vedi UC10).</li> </ol>
<b>Alternative Flow</b>	<ol style="list-style-type: none"> <li>2a. Se esiste già un invito attivo per lo stesso utente, il sistema non genera un nuovo codice e mostra il messaggio "The invitation has already been sent" (Test #).</li> <li>4a. Se il link scade o viene revocato dall'Admin, il sistema mostra "The invitation has expired" (Test #).</li> </ol>

Tabella 9: UC9 – Invite a New Member

UC10 – Manage Existing Members	
<b>Attore</b>	Admin
<b>Livello</b>	User Goal
<b>Pre-condizioni</b>	L'Admin è autenticato. Esiste almeno un gruppo attivo con uno o più membri.
<b>Basic Flow</b>	<ol style="list-style-type: none"> <li>1. L'Admin apre la sezione "Members Page".</li> <li>2. Il sistema mostra l'elenco dei membri con stato ("Active" / "Waiting") e saldo corrente.</li> <li>3. L'Admin può: <ul style="list-style-type: none"> <li>– Approvare/rifiutare membri in attesa</li> <li>– Rimuovere un membro esistente</li> <li>– Modificare i permessi</li> </ul> </li> <li>4. Il sistema aggiorna automaticamente lo stato del gruppo, i saldi e la cronologia.</li> <li>5. L'Admin riceve conferma delle modifiche effettuate.</li> </ol>
<b>Alternative Flow</b>	<ol style="list-style-type: none"> <li>3a. Se l'Admin tenta di rimuovere un membro con debiti aperti, il sistema blocca l'operazione e mostra il messaggio "The member cannot be removed" (Test #).</li> <li>3b. L'approvazione non va a buon fine per link scaduto: il sistema notifica errore "The invitation is no longer valid" (Test #).</li> </ol>

Tabella 10: UC10 – Manage Existing Members

UC11 – Edit/Delete Expense	
<b>Attore</b>	Admin
<b>Livello</b>	User Goal
<b>Pre-condizioni</b>	Esiste almeno una spesa registrata.
<b>Basic Flow</b>	<ol style="list-style-type: none"> <li>1. L'Admin accede alla sezione "Expense page".</li> <li>2. Seleziona la spesa da modificare o eliminare.</li> <li>3. Il sistema valida la coerenza dei dati modificati.</li> <li>4. Se confermata, il sistema aggiorna i saldi del gruppo e la cronologia.</li> <li>5. Il sistema notifica tutti i membri del gruppo.</li> </ol>
<b>Alternative Flow</b>	<ol style="list-style-type: none"> <li>2a. Tentativo di modifica su spesa già saldata: il sistema mostra un messaggio di blocco.</li> <li>2b. L'Admin annulla l'operazione: nessuna modifica viene salvata.</li> </ol>

Tabella 11: UC11 – Edit/Delete Expense

UC12 – Configure Group Settings	
<b>Attore</b>	Admin
<b>Livello</b>	User Goal
<b>Pre-condizioni</b>	L'Admin è autenticato e ha creato il gruppo.
<b>Basic Flow</b>	<ol style="list-style-type: none"> <li>1. L'Admin apre la sezione "Group settings".</li> <li>2. Visualizza le attuali impostazioni (nome, descrizione, valuta, regole).</li> <li>3. Modifica uno o più campi: <ul style="list-style-type: none"> <li>– Group name and description</li> <li>– Currency</li> <li>– Expense splitting rules</li> <li>– Spending limits</li> <li>– Notification settings</li> </ul> </li> <li>4. Il sistema valida i dati inseriti.</li> <li>5. L'Admin conferma le modifiche.</li> <li>6. Il sistema salva le nuove impostazioni e le applica ai futuri calcoli di saldo.</li> </ol>
<b>Alternative Flow</b>	<ol style="list-style-type: none"> <li>3a. Errore di validazione (es. valuta non supportata).</li> <li>3b. L'Admin annulla l'operazione: nessuna modifica viene applicata.</li> </ol>

Tabella 12: UC12 – Configure Group Settings

## 2.3 Mockups

I mockup, generati con **Figma**, illustrano una possibile rappresentazione grafica dell'interfaccia utente, coerente con i requisiti funzionali definiti.

The dashboard displays a grid of six expense sharing groups:

- Trip to Rome** (Active): 5 members. Your Balance: You owe **€20.00**.
- Housemates** (Active): 4 members. Your Balance: You are owed **€50.00**.
- Weekend Getaway** (Active): 3 members. Your Balance: You owe **€15.50**.
- Dinner Club** (Settled): 6 members. Your Balance: Settled up.
- Office Lunch** (Active): 6 members. Your Balance: You are owed **€12.75**.
- Skiing Trip 2028** (Active): 7 members. Your Balance: You owe **€45.00**.

Navigation links at the top right include: Dashboard, My Groups, Profile, Join Group, and Create New Group.

Figura 5: Mockup #1 – User Dashboard

The group details page for "Trip to Rome" (5 members, created on Jan 20, 2026) shows:

- Total Group Spending: **€887.25**
- Your Balance: **-€20.00** (You owe)
- Active Members: **5**

The "Expenses" section lists the following transactions:

DATE	DESCRIPTION	CATEGORY	Paid By	AMOUNT
Jan 28, 2026	Colosseum Tickets	⌚ Other	SC Sarah Chen	€84.00
Jan 27, 2026	Dinner at Trattoria	🍴 Food	MR Marco Rossi	€125.50
Jan 27, 2026	Taxi to Hotel	🚗 Transport	EW Emma Wilson	€22.00
Jan 26, 2026	Airbnb Payment	🏨 Accommodation	SC Sarah Chen	€450.00
Jan 26, 2026	Lunch at Cafe	🍴 Food	JD John Davis	€68.00
Jan 25, 2026	Grocery Shopping	🛒 Shopping	EW Emma Wilson	€45.75

Navigation links at the top right include: Dashboard, My Groups, Profile, Members, and Add Expense.

Figura 6: Mockup #2 – Group Details Page

**Add New Expense**

Amount	<input type="text" value="0.00"/> €
Description	<input type="text" value="e.g., Dinner at restaurant"/>
Category	<input type="text" value="Food &amp; Dining"/>
Paid By	<input type="text" value="You"/>
Split With	<input type="button" value="Deselect All"/>
<input checked="" type="checkbox"/> <b>YO</b> You €0.00 <input checked="" type="checkbox"/> <b>SC</b> Sarah Chen €0.00 <input checked="" type="checkbox"/> <b>MR</b> Marco Rossi €0.00 <input checked="" type="checkbox"/> <b>EW</b> Emma Wilson €0.00 <input checked="" type="checkbox"/> <b>JD</b> John Davis €0.00	
<input type="button" value="Cancel"/> <input type="button" value="Save Expense"/>	

Figura 7: Mockup #3 – Add Expense Modal

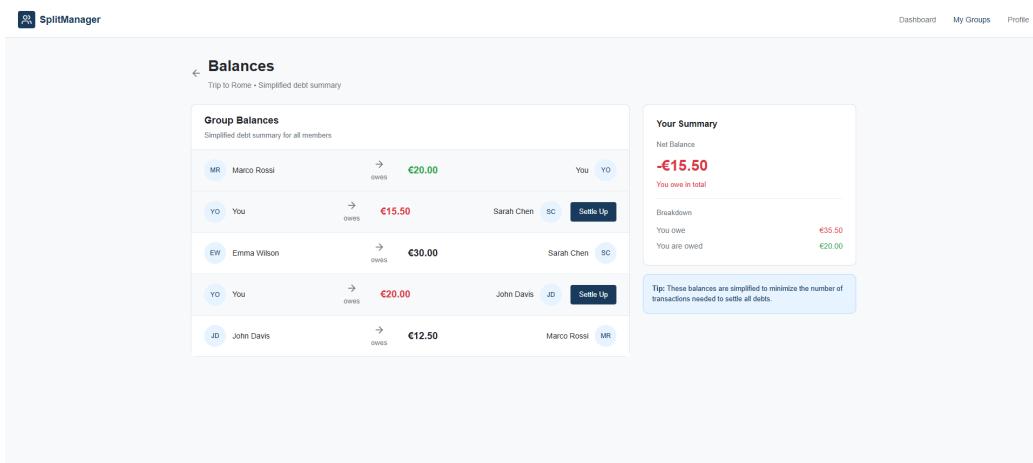


Figura 8: Mockup #4 – Balances & Settlement

## 2.4 Page Navigation Diagram

Il diagramma di navigazione descrive il flusso tra le diverse pagine dell'applicazione, evidenziando le transizioni possibili in base alle azioni dell'utente.

Come evidenziato nel diagramma (**Figura 9**), alcune pagine e azioni, come la gestione delle impostazioni di gruppo, la modifica/eliminazione delle spese e la gestione dei membri o inviti, sono riservate **solo** agli amministratori del gruppo. Questa restrizione visiva e funzionale garantisce che le operazioni critiche o potenzialmente distruttive non siano esposte ai membri standard, migliorando la sicurezza e prevenendo azioni accidentali.

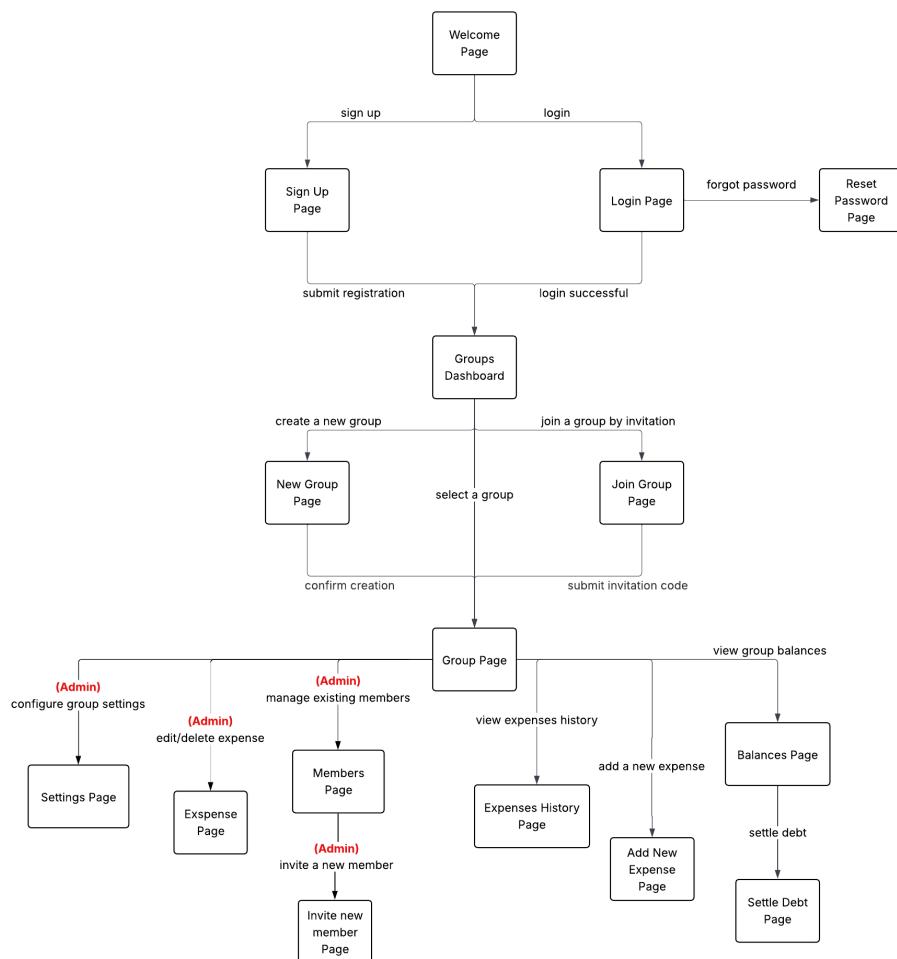


Figura 9: Page Navigation Diagram

# 3 Progettazione

## 3.1 Package Diagram

Il Package Diagram rappresenta l'organizzazione modulare del sistema e le dipendenze tra i principali sottosistemi.

I package principali sono:

- **Domain Model:** contiene le entità e le enumerazioni che modellano i concetti fondamentali del dominio (User, Group, Membership, Expense, Settlement, Balance). Questo livello definisce i comportamenti propri delle entità e le regole che ne governano lo stato, rimanendo indipendente dagli altri livelli applicativi.

Il Domain Model è stato a sua volta suddiviso in 3 sotto-package:

- **registry:** contiene le entità relative a utenti e gruppi
- **accounting:** contiene le entità relative alla contabilità
- **events:** contiene l'infrastruttura per il pattern Observer

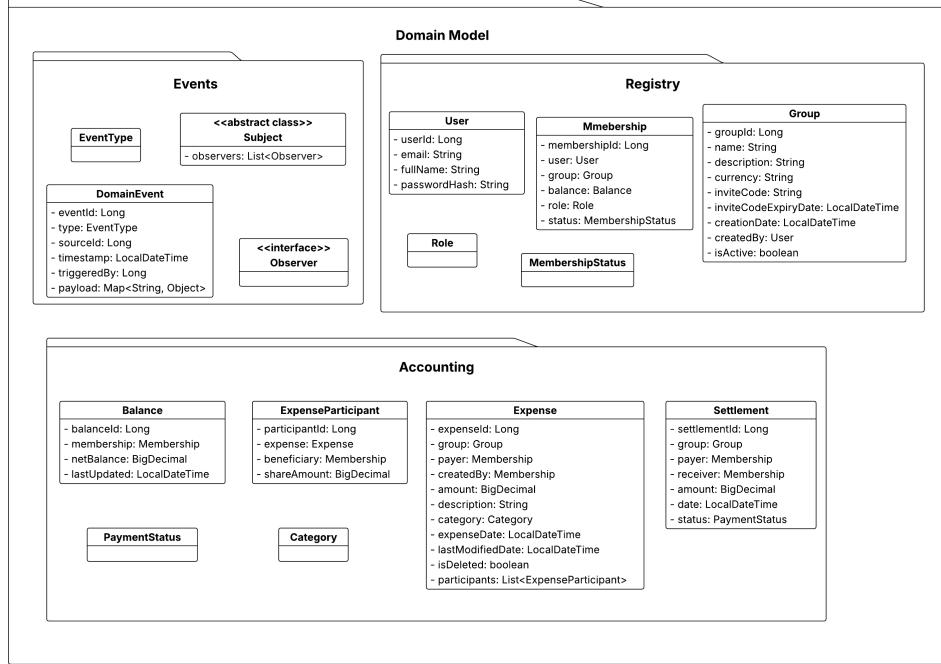


Figura 10: Suddivisione del Domain Model in sotto-package

- **Data Access Layer (ORM)**: gestisce la persistenza dei dati tramite il pattern DAO, isolando il resto dell'applicazione dai dettagli di accesso al database.
- **Service Layer**: implementa i casi d'uso applicativi coordinando più entità di dominio e interagendo con i DAO. È responsabile della gestione delle transazioni e del wiring degli Observer.
- **Controller Layer**: rappresenta il punto di accesso ai casi d'uso dal lato CLI. I controller validano le richieste e delegano la logica applicativa ai Service.
- **Util**: contiene componenti di supporto riutilizzabili. Include `PasswordHasher`, utilizzato per la gestione delle password.

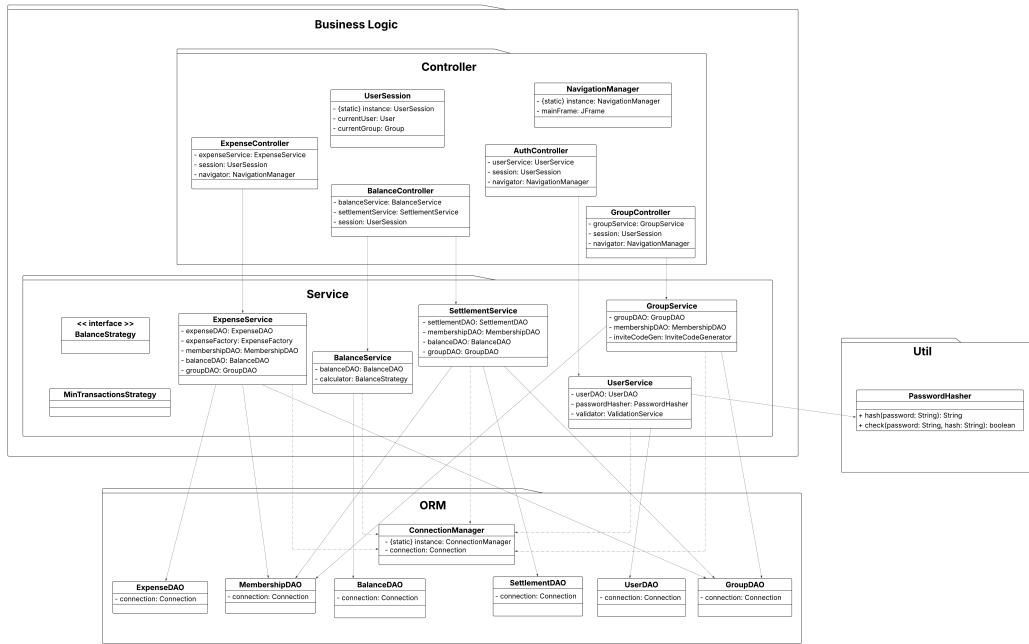


Figura 11: Dipendenze tra i package Controller, Service, ORM e Util

- **Exception**: raccoglie le eccezioni applicative personalizzate (`DomainException`, `DAOException`, `UnauthorizedException`, `EntityNotFoundException`), permettendo una gestione strutturata degli errori nei vari layer.

Le dipendenze seguono una direzione controllata: i Controller dipendono dai Service, i Service dipendono da Domain e DAO, mentre il Domain Model non dipende dagli altri layer. Le eccezioni sono condivise tra i layer per rappresentare condizioni di errore applicativo.

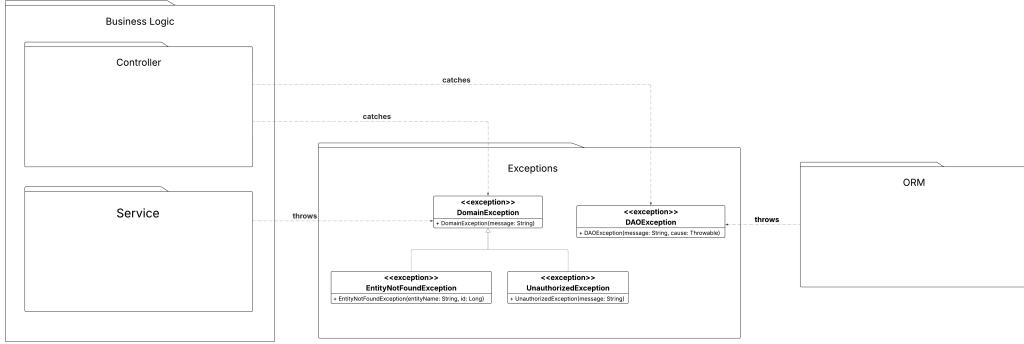


Figura 12: Exceptions Package

## 3.2 Class Diagrams

In questa sezione viene illustrata la struttura delle classi, evidenziando le relazioni tra le entità e l'applicazione dei design pattern.

### 3.2.1 Domain Model

Il diagramma delle classi del Domain Model descrive la logica di business alla base dell'applicazione. Come si è accennato, le classi sono state suddivise in 3 sotto-package all'interno del package Domain Model:

- 1. Events (Pattern Observer)** Questo package implementa il pattern Observer per gestire la propagazione dei cambiamenti di stato all'interno del Domain Model, assicurando che quando avviene un'azione (es. una spesa viene creata), le altre entità interessate (es. i saldi dei membri) vengano aggiornate automaticamente.
  - **Subject (Classe Astratta)**: Definisce il contratto per le entità che possono generare eventi (come Group, Expense, Settlement). Gestisce una lista transient di observer e fornisce metodi per l'aggancio (attach) e la notifica (notifyObservers).
  - **Observer (Interfaccia)**: Definita per le entità che devono reagire ai cambiamenti. Nel sistema, la classe Membership implementa questa interfaccia per aggiornare il proprio Balance in risposta agli eventi di dominio.
  - **DomainEvent (Value Object)**: Incapsula i dettagli di un cambiamento di stato, inclusi l'identificativo della sorgente (sourceId), il timestamp, l'utente che ha scatenato l'azione (triggeredBy) e un payload flessibile sotto forma di mappa.

- **EventType (Enumeration)**: Cataloga in modo esaustivo gli eventi possibili, distinguendo tra operazioni amministrative (es. MEMBER\_JOINED), contabili (EXPENSE\_CREATED) e di conguaglio (SETTLEMENT\_CONFIRMED)

## 2. Registry (Gestione Utenti e Gruppi)

- **User (Entity)**: Rappresenta l'attore principale del sistema, identificato univocamente da email e dotato di logica per la verifica delle credenziali.
- **Group (Entity, Subject)**: L'entità centrale che aggrega membri, spese e saldi. Gestisce i codici di invito e lo stato di attività. Estende Subject per notificare modifiche alla sua struttura.
- **Membership (Entity, Observer)**: Classe associativa che lega un User e un Group. Definisce il ruolo (ADMIN, MEMBER) e lo stato (ACTIVE, WAITING\_ACCEPTANCE, REMOVED). Implementa Observer per ricalcolare i debiti/crediti ogni volta che una spesa viene aggiunta o modificata nel gruppo di appartenenza.

## 3. Accounting (Contabilità)

- **Expense (Entity, Subject)**: Modella una spesa sostenuta da un membro (payer), include dettagli come l'importo (amount), la categoria e la data. Essendo un Subject, notifica gli osservatori (le Membership) affinché possano aggiornare i propri saldi netti.
- **ExpenseParticipant (Entity)**: Definisce quanto ogni beneficiario deve per una specifica spesa. Esiste in una relazione di composizione con Expense.
- **Balance (Entity)**: Ogni Membership ha un proprio Balance che traccia il debito/credito netto (netBalance) in tempo reale. Fornisce metodi per l'incremento o decremento del saldo e per la verifica dello stato di pareggio (isSettled).
- **Settlement (Entity, Subject)**: Rappresenta un'operazione di rimborso tra due membri (payer e receiver) per azzerare un debito. Include una gestione degli stati tramite PaymentStatus (PENDING, COMPLETED, REJECTED).

**Relazioni e Vincoli di Integrità** Il diagramma evidenzia legami strutturali forti che guidano la persistenza e il ciclo di vita degli oggetti:

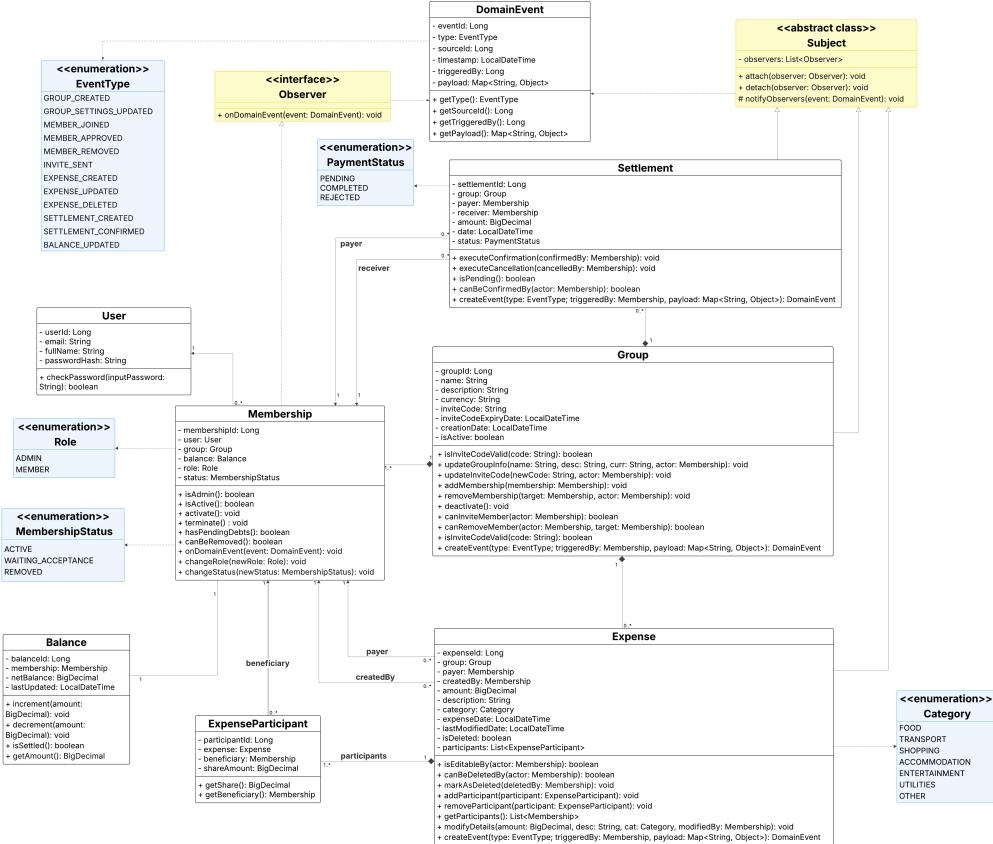


Figura 13: Domain Model Class Diagram

- Composizione:** Group esercita una composizione (1:N) su Membership, Expense e Settlement, indicando che queste entità non hanno ragione di esistere al di fuori del contesto del gruppo. Ogni Membership ha esattamente un Balance associato (1:1).
- Associazioni:** Un User può avere più Membership (partecipare a più gruppi), mentre una Expense ha un singolo payer (Membership) ma molti participants. Le spese e i pareggi puntano a istanze di Membership per identificare gli attori finanziari coinvolti, garantendo l'integrità referenziale all'interno del database.

### 3.2.2 ORM

Il diagramma delle classi dell'ORM descrive il livello di persistenza del sistema, responsabile della traduzione tra oggetti del Domain Model e dati memorizzati nel database relazionale. L'accesso ai dati è organizzato secondo il pattern

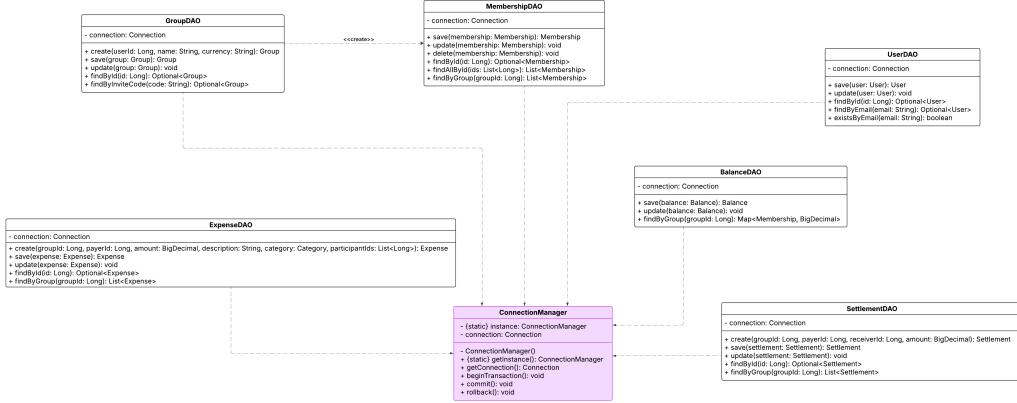


Figura 14: ORM Class Diagram

Data Access Object (DAO), che incapsula i dettagli delle query SQL fornendo un’interfaccia orientata agli oggetti al Service Layer.

## Componenti principali:

### 1. ConnectionManager

La gestione delle connessioni al database è modellata tramite la classe **ConnectionManager**, rappresentata come Singleton. Essa fornisce l’accesso alla connessione JDBC e i metodi per la gestione delle transazioni: **beginTransaction()**, **commit()** e **rollback()**. Tutte le classi DAO convergono verso il **ConnectionManager**, da cui ottengono l’oggetto **Connection** necessario per eseguire le operazioni di persistenza.

### 2. Concrete DAOs

Per ciascuna entità principale del dominio è previsto un DAO dedicato (**UserDAO**, **GroupDAO**, **MembershipDAO**, **ExpenseDAO**, **BalanceDAO**, **SettlementDAO**). Ogni DAO espone metodi per il salvataggio, l’aggiornamento e il recupero delle entità. Come evidenziato nel diagramma, i DAO includono metodi specifici per il dominio, quali **findByInviteCode** in **GroupDAO** o **findByGroup** in **ExpenseDAO** per recuperare le spese di un contesto specifico.

I DAO hanno la responsabilità architetturale di istanziare gli oggetti del Domain Model (**User**, **Group**, ecc.), mappando i risultati delle query (**ResultSet**) in oggetti Java utilizzabili dai layer superiori.

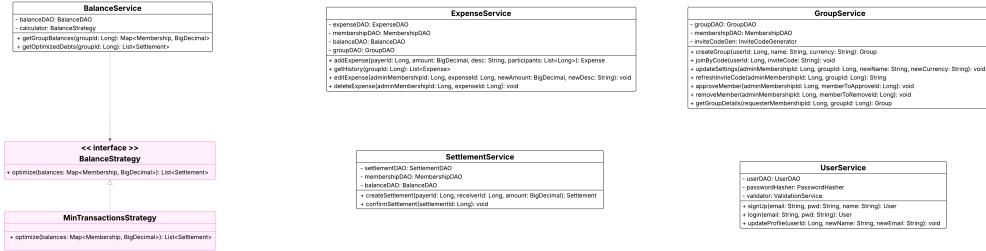


Figura 15: Service Layer Class Diagram

**Gestione delle transazioni** Il controllo delle transazioni è delegato al **Service Layer**. I Service aprono una transazione tramite il **ConnectionManager**, coordinano l'esecuzione dei vari DAO e infine consolidano (commit) o annullano (rollback) l'intero blocco di modifiche.

Questo approccio consente di eseguire in modo atomico operazioni che coinvolgono più entità. Ad esempio, la creazione di una nuova spesa richiede sia l'inserimento dell'**Expense** nel database, sia l'aggiornamento dei **Balance** di tutti i membri coinvolti; queste operazioni devono essere eseguite all'interno della stessa transazione per garantire la consistenza dei dati.

**Gestione delle eccezioni** I DAO intercettano le eccezioni di basso livello (`SQLException`) e le incapsulano in `DAOException`. In questo modo i layer superiori rimangono indipendenti dai dettagli della tecnologia di persistenza e possono gestire gli errori a un livello più astratto.

### 3.2.3 Service Layer

Il Service Layer rappresenta il nucleo funzionale dell'applicazione. Le classi di questo livello orchestrano interi casi d'uso coordinando entità di dominio e componenti di persistenza. Ogni classe di servizio rappresenta un'area funzionale ben definita del sistema:

- **UserService**: gestisce registrazione, autenticazione e aggiornamento del profilo utente, utilizzando il **PasswordHasher** per la sicurezza delle credenziali.
- **GroupService**: gestisce la creazione dei gruppi, la generazione dei codici invito, l'ingresso nei gruppi e le operazioni amministrative sui membri.
- **ExpenseService**: coordina la creazione, modifica e cancellazione delle spese, assicurando la coerenza dei dati contabili.

- **SettlementService**: gestisce i rimborsi tra membri e la loro conferma.
- **BalanceService**: fornisce funzionalità di consultazione dei saldi e di ottimizzazione dei debiti.

Oltre a coordinare operazioni che coinvolgono più entità e più DAO all'interno della stessa operazione logica, i Service sono responsabili della **gestione transazionale**: definiscono i confini delle transazioni atomiche interagendo con il **ConnectionManager** (commit/rollback).

Il Service Layer è anche responsabile della gestione del ciclo di vita del pattern Observer previsto nel Domain Model. Poiché le liste di Observer associate ai Subject (Expense, Settlement, Group) non vengono persistite nel database, prima di eseguire un'operazione che comporta un cambiamento di stato, il Service Layer ricostruisce dinamicamente (**Wiring**) queste dipendenze a runtime per garantire il corretto funzionamento del meccanismo di notifica.

### **Applicazione dello Strategy Pattern per l'Ottimizzazione dei Debiti**

Quando gli utenti registrano numerose spese incrociate, il sistema deve capire chi deve pagare chi, cercando di ridurre al minimo il numero di bonifici necessari per azzerare i debiti di tutti. Per risolvere questo problema, è stato implementato un algoritmo di ottimizzazione dei debiti all'interno del **BalanceService** che delega la complessa logica di calcolo a un componente esterno, sfruttando il pattern comportamentale **Strategy**. Il pattern Strategy permette di separare "chi gestisce i dati" da "chi esegue i calcoli", strutturando la soluzione in tre elementi chiave:

1. **Il Contesto (BalanceService)**: Agisce come un coordinatore, si occupa di recuperare i saldi aggiornati dal database e passarli all'algoritmo di calcolo, senza preoccuparsi di come questo funzioni internamente.
2. **L'Interfaccia (BalanceStrategy)**: Definisce il contratto standard di comunicazione, espone un unico metodo, `optimize`, che prende in input la mappa dei saldi dei membri (`Map<Membership, BigDecimal>`) e garantisce di restituire una lista di transazioni da effettuare (`List<Settlement>`).
3. **La Strategia Concreta (MinTransactionsStrategy)**: È la classe che implementa fisicamente l'interfaccia e contiene il vero e proprio algoritmo matematico progettato per minimizzare gli scambi di denaro.

Questa architettura garantisce elevata flessibilità e manutenibilità, rispettando il principio **Open/Closed** (Aperto alle estensioni, Chiuso alle modifiche), qualora in futuro si rendesse necessario introdurre un nuovo criterio per il calcolo dei rimborsi (ad esempio, un algoritmo che arrotonda le cifre per

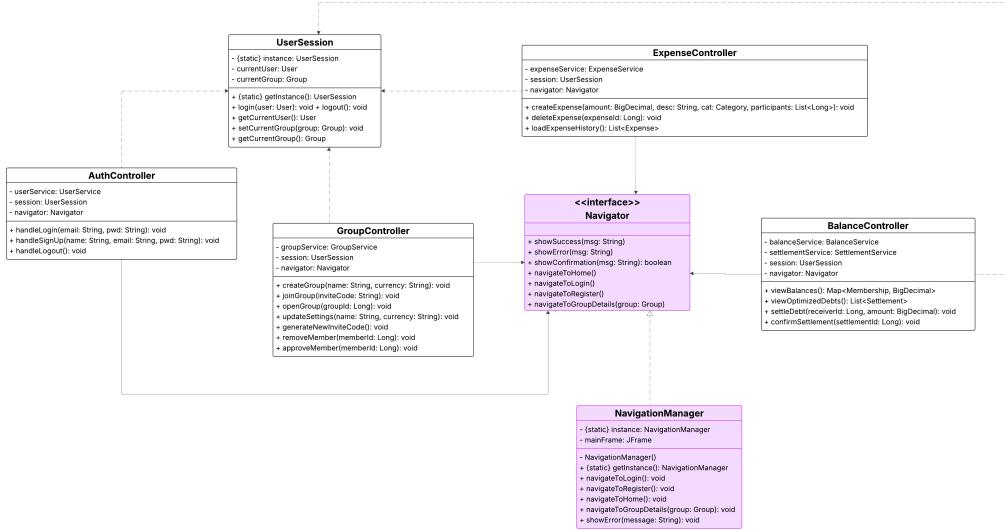


Figura 16: Controller Layer Class Diagram

agevolare lo scambio di contanti), sarà sufficiente sviluppare una nuova classe che implementi **BalanceStrategy**, lasciando invariato il codice del **BalanceService**.

### 3.2.4 Controller Layer

Il Controller Layer rappresenta il punto di accesso ai casi d'uso dal lato CLI. I Controller (**AuthController**, **GroupController**, **ExpenseController**, **BalanceController**) non contengono business logic complessa, ma svolgono un ruolo di coordinamento tra interfaccia utente e service: ricevono le richieste dell'utente, ne verificano la validità formale e delegano l'elaborazione ai servizi appropriati.

Per supportare il funzionamento dei Controller, come illustrato in Figura 16, il sistema include due componenti infrastrutturali modellati come Singleton, scelti per garantire un punto di accesso globale controllato:

- **UserSession**: gestisce il contesto e lo stato globale dell'applicazione. Conserva i riferimenti all'utente attualmente autenticato (**currentUser**) e al gruppo selezionato (**currentGroup**). Questo approccio centralizzato evita di dover passare continuamente questi parametri a ogni singola invocazione dei metodi del Controller o del Service.
- **NavigationManager**: implementa l'interfaccia **Navigator** e centralizza il flusso di transizione tra i menu della CLI. Questa struttura applica

il *Dependency Inversion Principle*: i controller dipendono unicamente dall’astrazione e non dall’implementazione concreta, garantendo il totale disaccoppiamento della logica applicativa dall’input/output su console. Questo è fondamentale per consentire l’esecuzione isolata dei test automatizzati (questo aspetto verrà approfondito nella **Sezione 5.3**).

### 3.3 Database

La progettazione del database è stata effettuata in continuità con il Domain Model, garantendo allineamento tra entità software e strutture relazionali. Le relazioni, le cardinalità e i vincoli di integrità presenti nel modello ER riflettono quelli definiti nel diagramma delle classi.

#### 3.3.1 Modello ER

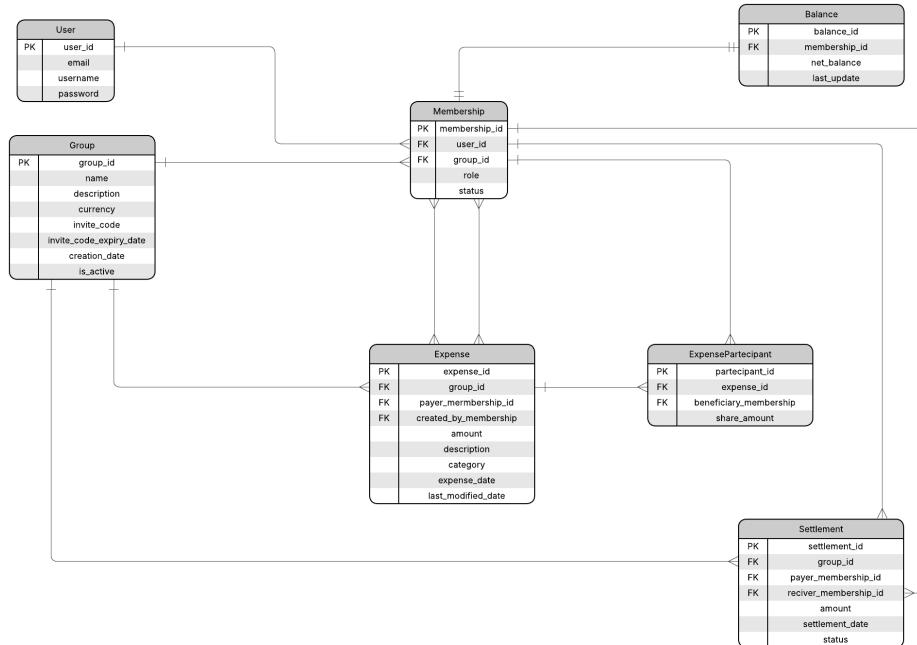


Figura 17: Modello ER

### 3.3.2 Schema Relazionale

Dalla traduzione del modello ER si ottiene il seguente schema relazionale. Le chiavi primarie (PK) sono sottolineate, mentre le chiavi esterne (FK) mantengono il nome dell'attributo referenziato.

**User** (user\_id, email, full\_name, password)

**Group** (group\_id, name, description, currency, invite\_code, invite\_code\_expiry\_date, creation\_date, created\_by\_user\_id, is\_active)

**Membership** (membership\_id, user\_id, group\_id, role, status)

**Expense** (expense\_id, group\_id, payer\_membership\_id, created\_by\_membership, amount, description, category, expense\_date, last\_modified\_date, is\_deleted)

**ExpenseParticipant** (participant\_id, expense\_id, beneficiary\_membership\_id, share\_amount)

**Balance** (balance\_id, membership\_id, net\_balance, last\_updated)

**Settlement** (settlement\_id, group\_id, payer\_membership\_id, receiver\_membership\_id, amount, settlement\_date, status)

## **4 Implementazione**

L'implementazione del sistema è stata realizzata seguendo le scelte architettoniche definite in fase di progettazione, con l'obiettivo di mantenere una chiara separazione delle responsabilità tra i diversi livelli applicativi.

## 5 Test

Il sistema è stato testato seguendo il modello della **Test Pyramid**, coprendo tre livelli di granularità: Unit (White-Box), Integration (Grey-Box) e Functional (Black-Box).

### 5.1 Test Strutturali (White-Box)

Questo livello è alla base della piramide e verifica le classi del **Domain Model** in isolamento. I test verificano che le invarianti di classe siano rispettate e che la logica di business risponda correttamente, garantendo che ogni componente si auto-protegga da input non validi e mantenga uno stato consistente.

Le principali suite di test implementate sono:

- **BalanceTest:** verifica le invarianti della classe `Balance`, in particolare la corretta associazione a una membership valida (`membership != null`), la correttezza delle operazioni sui saldi, come `increment()` e `decrement()`, e la coerenza dello stato "settled".

```
@Test
void settle_resetsToZero_andIsSettled() {
    Balance b = new Balance(null, memberA);
    b.increment(new BigDecimal("9.99"));
    assertFalse(b.isSettled());
    b.settle();
    assertTrue(b.isSettled());
    assertEquals(BigDecimal.ZERO.setScale(2), b.getAmount());
}
```

Figura 18: Snippet 1 - BalanceTest

- **ExpenseTest:** verifica validazioni sull'importo (`amount > 0`), le regole di autorizzazione che definiscono i permessi di modifica ed eliminazione delle spese e il meccanismo di soft delete (le spese non vengono fisicamente eliminate dal database, ma marcate come `isDeleted = true`).
- **MembershipTest:** verifica le invarianti legate allo stato delle membership e la loro relazione con il `Balance` associato. In particolare, viene testata la coerenza tra la presenza di debiti (`hasPendingDebts()`) e la possibilità di rimuovere un membro dal gruppo (`canBeRemoved()`) per garantire il corretto comportamento dello UC10.

```

@Test
void testConstructor_WithNegativeAmount_ThrowsException() {
    // Verifica che l'entità si auto-protectta
    assertThrows(
        DomainException.class,
        () -> new Expense(
            1L, group, creator, creator,
            new BigDecimal("-100.00"), // Importo negativo
            "Invalid Expense",
            Category.FOOD,
            LocalDateTime.now()
        )
    );
}

```

Figura 19: Snippet 2 - ExpenseTest

```

@Test
void canBeRemoved_reflectsPendingDebts() {
    Membership m = new Membership(null, userOther, group, Role.MEMBER);
    assertTrue(m.canBeRemoved()); // senza balance

    Balance b = new Balance(null, m);
    b.increment(new BigDecimal("1.00")); // crea debito
    m.setBalance(b);

    assertFalse(m.canBeRemoved()); // ha debiti, non può essere rimossa
}

```

Figura 20: Snippet 3 - MembershipTest

## 5.2 Test di Integrazione (Grey-Box)

### 5.3 Test Funzionali (Black-Box)

Questo livello verifica il comportamento del sistema dal punto di vista dell'utente, esercitando i casi d'uso attraverso il Controller layer utilizzato come API.

È stata implementata una suite completa, E2ETest, che simula uno scenario di utilizzo tipico (dalla creazione del gruppo all'estinzione dei debiti) attraverso il caso di test `testCompleteUserFlow`.

#### 5.3.1 Scenario di Test

Due utenti, Alice e Bob, utilizzano il sistema per gestire una spesa comune.

1. Alice si registra al sistema e crea un nuovo gruppo "Vacation". Il sistema assegna automaticamente ad Alice il ruolo di ADMIN e genera

un codice invito.

2. Bob utilizza il codice invito generato da Alice per unirsi al gruppo. Inizialmente la sua membership è in stato WAITING\_ACCEPTANCE, finché Alice non lo approva.
3. Alice registra una spesa di 100€ per l'hotel, suddivisa equamente con Bob. Il sistema calcola automaticamente i saldi dei membri in base alla ripartizione della spesa.
4. Bob crea un settlement per saldare il suo debito di 50€ verso Alice. Il settlement parte in stato PENDING e richiede la conferma di Alice (in quanto ricevente del pagamento) per essere completato. Solo dopo la conferma i balance vengono aggiornati. Questo meccanismo di "doppia conferma" protegge entrambe le parti.
5. Dopo la conferma del settlement, il sistema deve aver aggiornato i balance a zero per entrambi i membri e il gruppo deve risultare completamente saldato.

### 5.3.2 Aspetti Architetturali

Il test E2E è eseguito senza coinvolgere direttamente la CLI interattiva. Per consentire ciò è stato applicato il *Dependency Inversion Principle*, facendo dipendere i controller dall'interfaccia **Navigator** anziché dall'implementazione concreta che gestisce i menu.

Questa architettura permette di iniettare nel test uno Stub di **Navigator** che intercetta le richieste di navigazione senza bloccare l'esecuzione in attesa di input da tastiera. In produzione, invece, viene utilizzato il **NavigationManager** standard che gestisce il flusso della CLI.

In questo modo i casi d'uso possono essere testati automaticamente, mantenendo la logica applicativa isolata dal layer di input/output su console. La CLI gestisce esclusivamente l'interazione con l'utente, delegando la logica applicativa ai Controller, mentre i test esercitano direttamente tali controller.

```

| static class StubNavigator implements Navigator {
|     public boolean hasError = false;
|     public String lastMessage = "";
|     public String currentPage = "NONE";
|
|     @Override
|     public void showSuccess(String message) {
|         this.lastMessage = message;
|         this.hasError = false;
|     }
|
|     @Override
|     public void showError(String message) {
|         this.lastMessage = message;
|         this.hasError = true;
|     }
|
|     @Override
|     public boolean showConfirmation(String message) {
|         return true;
|     }
|
|     @Override
|     public void navigateToLogin() {
|         this.currentPage = "LOGIN";
|     }
|
|     @Override
|     public void navigateToRegister() {
|         this.currentPage = "REGISTER";
|     }
|
|     @Override
|     public void navigateToHome() {
|         this.currentPage = "HOME";
|     }
|
|     @Override
|     public void navigateToGroupDetails(Group group) {
|         this.currentPage = "GROUP_DETAILS";
|     }
| }

```

Figura 21: Snippet 4 - StubNavigator