Università degli Studi di Firenze

Dipartimento di Ingegneria dell'Informazione

SplitManager

Sistema software per la gestione delle spese condivise

Autori	**Matricole**
Carmen Possidente	7115970
Matteo Gerlotti	7025024
Roberta Donato	7113502

Corso: Ingegneria del Software
Docente: Prof. Enrico Vicario

Anno Accademico 2025–2026

Indice

Elenco delle figure

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.

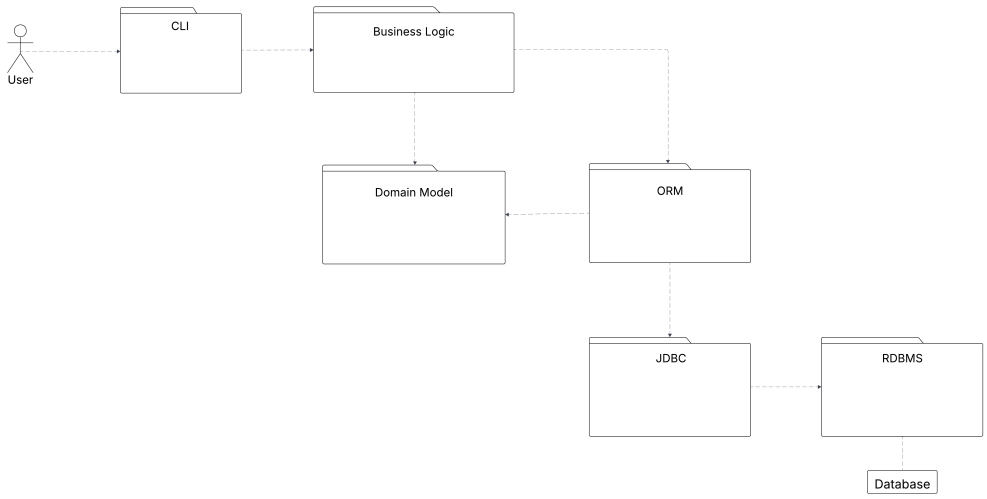

Figura 1: Panoramica dell'architettura logica a livelli del sistema.

1.3 Tecnologie e Strumenti

Il sistema è sviluppato in **Java**, con **Maven** come strumento di build e gestione delle dipendenze. Per la persistenza dei dati è stato utilizzato un approccio basato su JDBC interfacciato con un database relazionale in-memory **H2**, mentre per il testing è stato utilizzato **JUnit 5**.

Tutti i diagrammi sono stati realizzati con **Lucidchart**, mentre i mockup sono stati prodotti con **Figma**, avvalendosi della funzionalità di intelligenza artificiale generativa integrata nella piattaforma per la generazione delle schermate.

Il codice sorgente è disponibile al seguente repository:
`https://github.com/mgerlo/SWE_project`

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.

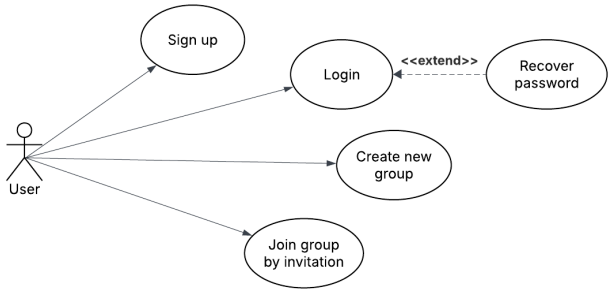

Figura 2: Use Case Diagram - User

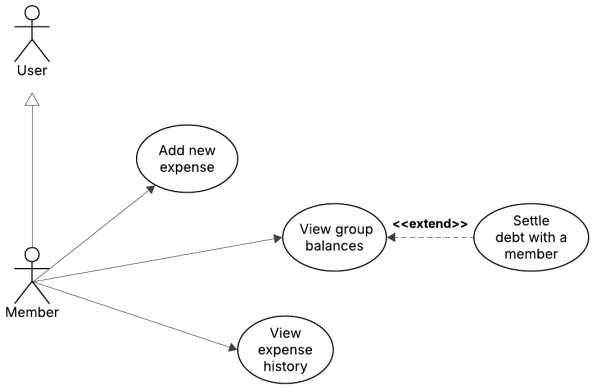

Figura 3: Use Case Diagram - Member

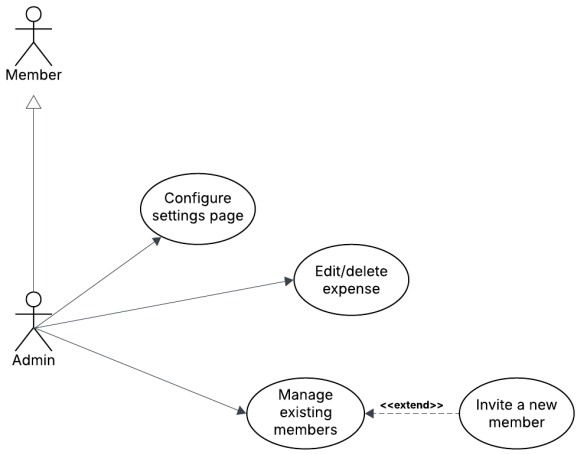

Figura 4: Use Case Diagram - Admin

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	
Attore	User
Livello	Function
Pre-condizioni	L'utente non deve essere già registrato nel sistema.
Basic Flow	1. L'utente accede alla pagina iniziale di SplitManager. 2. Clicca su "Sign up". 3. Il sistema apre la pagina "Create Account". 4. L'utente inserisce i dati richiesti: – Full name – Email address – Password – Confirm password 5. Il sistema verifica la validità dei dati (formato email, corrispondenza password). 6. Il sistema registra il nuovo utente nel database. 7. Il sistema mostra un messaggio di conferma e reindirizza alla pagina di login (*Test*).
Alternative Flow	5a. Se l'email è già registrata, il sistema mostra un messaggio di errore "Account already exists" (*Test*). 5b. Se i campi obbligatori non sono compilati o le password non coincidono, il sistema richiede la correzione (*Test*).

Tabella 1: UC1 – Sign Up

UC2 – Login	
Attore	User
Livello	Function
Pre-condizioni	L'utente deve essere già registrato nel sistema.
Basic Flow	1. L'utente accede alla pagina "Login". 2. Inserisce le proprie credenziali (email e password). 3. Clicca su "Login". 4. Il sistema verifica le credenziali nel database. 5. Se valide, il sistema apre la Home Page utente, dove può creare o unirsi a gruppi.
Alternative Flow	4a. Se l'email non è registrata, il sistema mostra un messaggio di errore (*Test*). Se la password non è corretta, mostra "Invalid credentials" e permette un nuovo tentativo (*Test*). 4b. Se l'utente dimentica la password, può cliccare su "Forgot password?" per avviare la procedura di recupero.

Tabella 2: UC2 – Login

UC3 – Create New Group	
Attore	User
Livello	User Goal
Pre-condizioni	L'utente deve essere autenticato (logged in).
Basic Flow	1. L'utente accede alla dashboard personale (*vedi Mockup #1*). 2. Clicca su "Create New Group". 3. Il sistema apre la pagina di configurazione gruppo. 4. L'utente inserisce i dettagli del gruppo: – Group name – Description (facoltativa) – Currency (selezionabile da dropdown) 5. Clicca su "Create". 6. Il sistema registra il nuovo gruppo, imposta l'utente come Admin, e apre la pagina del gruppo creato (*Test*).
Alternative Flow	5a. Se mancano dati obbligatori (es. nome gruppo), il sistema mostra un messaggio di errore "Please complete all required fields". 5b. Se si verifica un errore di rete o salvataggio, viene mostrato "Group creation failed. Try again later."

Tabella 3: UC3 – Create New Group

UC4 – Join Group by Invitation	
Attore	User
Livello	User Goal
Pre-condizioni	L'utente deve avere un account valido ed essere loggato. Deve aver ricevuto un link o un codice invito generato da un Admin.
Basic Flow	1. L'utente accede alla sezione "Join Group" dalla Dashboard (*vedi Mockup #1*). 2. Inserisce il codice di invito oppure clicca direttamente sul link ricevuto. 3. Il sistema verifica la validità del codice e l'esistenza del gruppo. 4. Se valido, il sistema aggiunge l'utente come Member del gruppo in stato `WAITING_ACCEPTANCE` (*Test*). 5. Il sistema mostra un messaggio di conferma e reindirizza alla pagina del gruppo (*vedi Mockup #2*).
Alternative Flow	3a. Se il codice è errato o scaduto, il sistema mostra "Invalid or expired invitation" (*Test*). 3b. Se l'utente è già membro del gruppo, il sistema mostra "You are already part of this group".

Tabella 4: UC4 – Join Group by Invitation

UC5 – Add New Expense	
Attore	Member
Livello	User Goal
Pre-condizioni	Il membro deve appartenere ad almeno un gruppo.
Basic Flow	1. Il membro accede alla pagina del gruppo. 2. Clicca sul pulsante "Add Expense" in alto a destra. 3. Il sistema apre la finestra modale di inserimento (*vedi Mockup #3*). 4. Il membro inserisce i dettagli della spesa: – Description – Amount – Category (tramite dropdown) – Paid by (tramite dropdown) – Split between (checkbox membri o "Select all") 5. Il sistema verifica la correttezza dei dati inseriti. 6. Il sistema registra la nuova spesa, aggiorna i saldi automaticamente tramite il Pattern Observer, e chiude il modale (*Test*).
Alternative Flow	4a. Se l'importo non è valido (negativo o zero), il sistema mostra un messaggio di errore (*Test*). 5a. Se i dati sono incompleti, il sistema richiede il completamento dei campi mancanti.

Tabella 5: UC5 – Add New Expense

UC6 – View Group Balances	
Attore	Member
Livello	User Goal
Pre-condizioni	Il membro deve appartenere ad almeno un gruppo.
Basic Flow	1. Il membro accede alla pagina del gruppo. 2. Seleziona la sezione "Balances" (*vedi Mockup #4*). 3. Il sistema mostra: – Totale delle spese del gruppo – Saldo individuale di ciascun membro (*Test*) – Lista di debiti/crediti reciproci ottimizzata da `MinTransactionsStrategy` 4. Il membro può visualizzare i dettagli delle transazioni passate.
Alternative Flow	3a. Se non ci sono spese registrate, il sistema mostra "No expenses yet" (*Test*).

Tabella 6: UC6 – View Group Balances

UC7 – View Expense History	
Attore	Member
Livello	User Goal
Pre-condizioni	Il membro appartiene ad un gruppo con almeno una spesa registrata.
Basic Flow	1. Il membro accede alla sezione "Expense History" del gruppo. 2. Il sistema mostra una lista cronologica delle spese con: – Data – Descrizione – Importo – Chi ha pagato – Tra chi è stata divisa 3. Il membro può filtrare o cercare spese specifiche per categoria, data o membro.
Alternative Flow	2a. Se non ci sono spese nel periodo selezionato, il sistema mostra "No expenses found".

Tabella 7: UC7 – View Expense History

UC8 – Settle Debt with a Member	
Attore	Member
Livello	User Goal
Pre-condizioni	Il membro ha un debito aperto verso un altro membro del gruppo.
Basic Flow	1. Il membro accede alla sezione "Balances" (*vedi Mockup #4*). 2. Seleziona il debito da saldare. 3. Il sistema mostra l'importo dovuto e chiede conferma del pagamento. 4. Il membro conferma. 5. Il sistema registra il pagamento in stato `PENDING` e aggiorna i saldi solo alla conferma definitiva del creditore.
Alternative Flow	4a. Se il membro annulla l'operazione, il sistema chiude la richiesta senza registrare il pagamento e i saldi rimangono invariati (*Test*). 4b. Se l'importo inserito supera il debito reale, il sistema blocca l'operazione (*Test*).

Tabella 8: UC8 – Settle Debt with a Member

UC9 – Invite a New Member	
Attore	Admin
Livello	User Goal
Pre-condizioni	L'Admin è autenticato e si trova nella pagina del gruppo.
Basic Flow	1. L'Admin accede alla sezione "Members Page" del gruppo. 2. Seleziona l'opzione "Invite a new member". 3. Il sistema genera un codice/link di invito univoco (*Test*). 4. L'Admin copia o invia il link ai potenziali membri tramite canali esterni. 5. Il sistema registra l'invito in stato "Waiting for acceptance". 6. Quando l'utente destinatario utilizza il link, il sistema valida il codice e invia la richiesta di ingresso al gruppo. 7. L'Admin riceve la notifica di nuova richiesta e potrà successivamente approvarla (vedi UC10).
Alternative Flow	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". 4a. Se il link scade o viene revocato dall'Admin, il sistema mostra "The invitation has expired". 5a. Se un membro non-admin tenta di generare un codice invito, il sistema rifiuta l'operazione (*Test*).

Tabella 9: UC9 – Invite a New Member

UC10 – Manage Existing Members	
Attore	Member (creatore della spesa) o Admin
Livello	User Goal
Pre-condizioni	L'Admin è autenticato. Esiste almeno un gruppo attivo con uno o più membri.
Basic Flow	1. L'Admin apre la sezione "Members Page". 2. Il sistema mostra l'elenco dei membri con stato ("Active"/"Waiting") e saldo corrente. 3. L'Admin può: – Approvare/rifiutare membri in attesa – Rimuovere un membro esistente – Modificare i permessi 4. Il sistema aggiorna automaticamente lo stato del gruppo, i saldi e la cronologia. 5. L'Admin riceve conferma delle modifiche effettuate.
Alternative Flow	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*). 3b. L'approvazione non va a buon fine per link scaduto: il sistema notifica errore "The invitation is no longer valid".

Tabella 10: UC10 – Manage Existing Members

UC11 – Edit/Delete Expense	
Attore	Admin
Livello	User Goal
Pre-condizioni	Il membro è autenticato ed è il creatore della spesa, oppure è l'Admin del gruppo.
Basic Flow	1. L'Admin accede alla sezione "Expense page". 2. Seleziona la spesa da modificare o eliminare. 3. Il sistema valida la coerenza dei dati modificati. 4. Se confermata, il sistema aggiorna i saldi del gruppo tramite il Pattern Observer (*Test*) e, in caso di eliminazione, li annulla completamente (*Test*). 5. Il sistema notifica tutti i membri del gruppo.
Alternative Flow	2a. Tentativo di modifica o eliminazione da parte di un membro non creatore della spesa: il sistema blocca l'operazione (*Test*). 2b. L'Admin annulla l'operazione: nessuna modifica viene salvata.

Tabella 11: UC11 – Edit/Delete Expense

UC12 – Configure Group Settings	
Attore	Admin
Livello	User Goal
Pre-condizioni	L'Admin è autenticato e ha creato il gruppo.
Basic Flow	1. L'Admin apre la sezione "Group settings". 2. Visualizza le attuali impostazioni (nome, descrizione, valuta, regole). 3. Modifica uno o più campi: – Group name and description – Currency – Expense splitting rules – Spending limits – Notification settings 4. Il sistema valida i dati inseriti. 5. L'Admin conferma le modifiche. 6. Il sistema salva le nuove impostazioni e le applica ai futuri calcoli di saldo.
Alternative Flow	3a. Errore di validazione (es. valuta non supportata o nome vuoto). 3b. Se un membro non-admin tenta di modificare le impostazioni, il sistema rifiuta l'operazione (*Test*). 3c. L'Admin annulla l'operazione: nessuna modifica viene applicata.

Tabella 12: UC12 – Configure Group Settings

2.3 Mockups

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

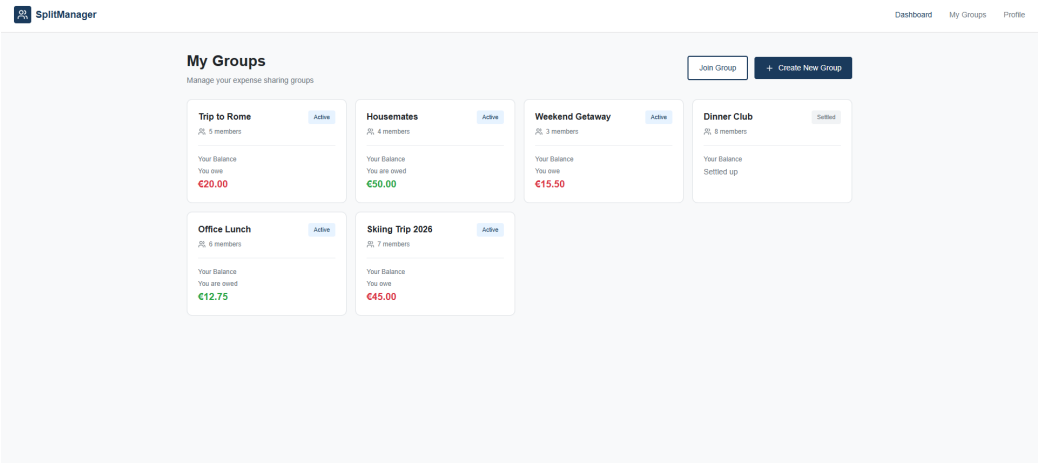

Figura 5: Mockup #1 – User Dashboard

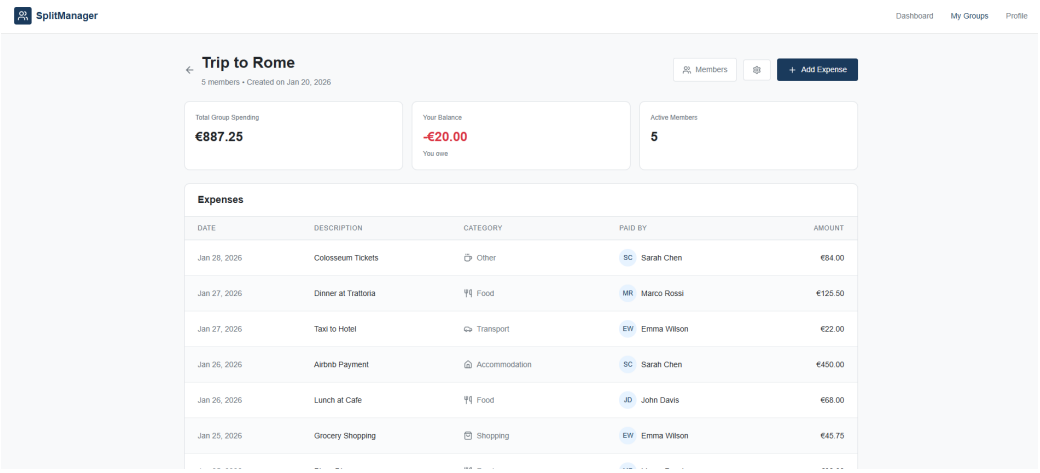

Figura 6: Mockup #2 – Group Details Page

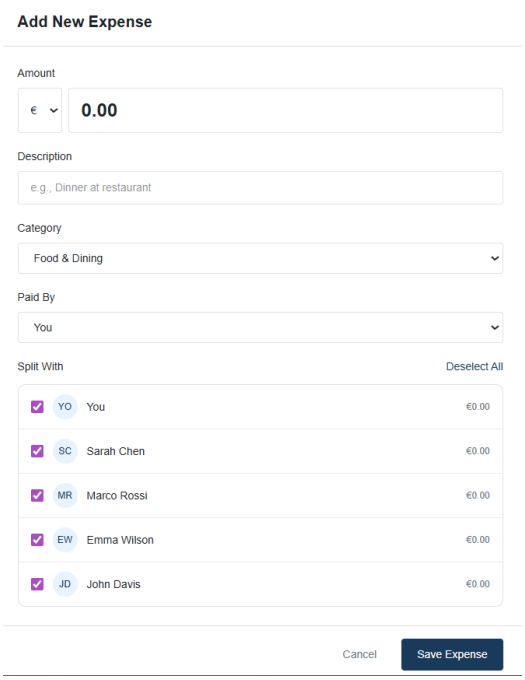

Figura 7: Mockup #3 – Add Expense Modal

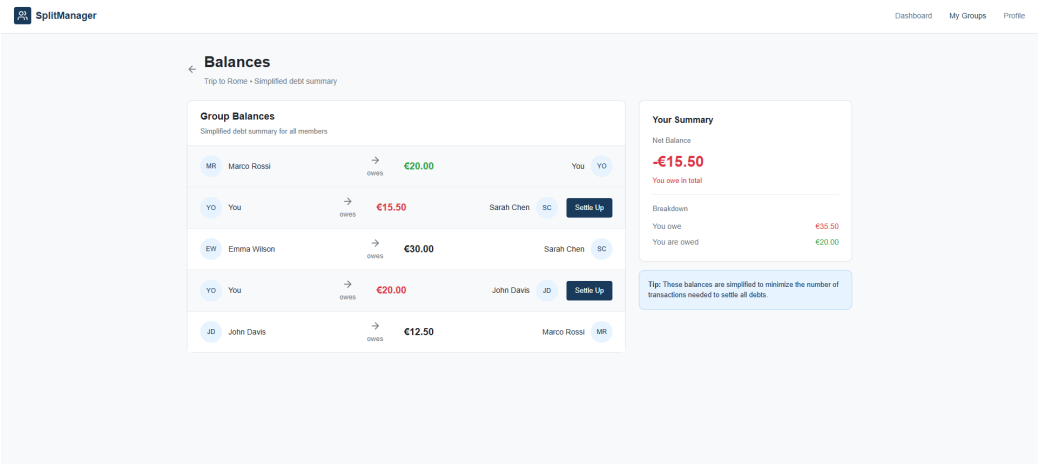

Figura 8: Mockup #4 – Balances & Settlement

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 e la gestione dei membri o inviti, sono riservate **solo** agli amministratori del gruppo. La modifica o eliminazione delle spese, invece, è permessa sia agli amministratori che ai membri che hanno originariamente creato la spesa. 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.

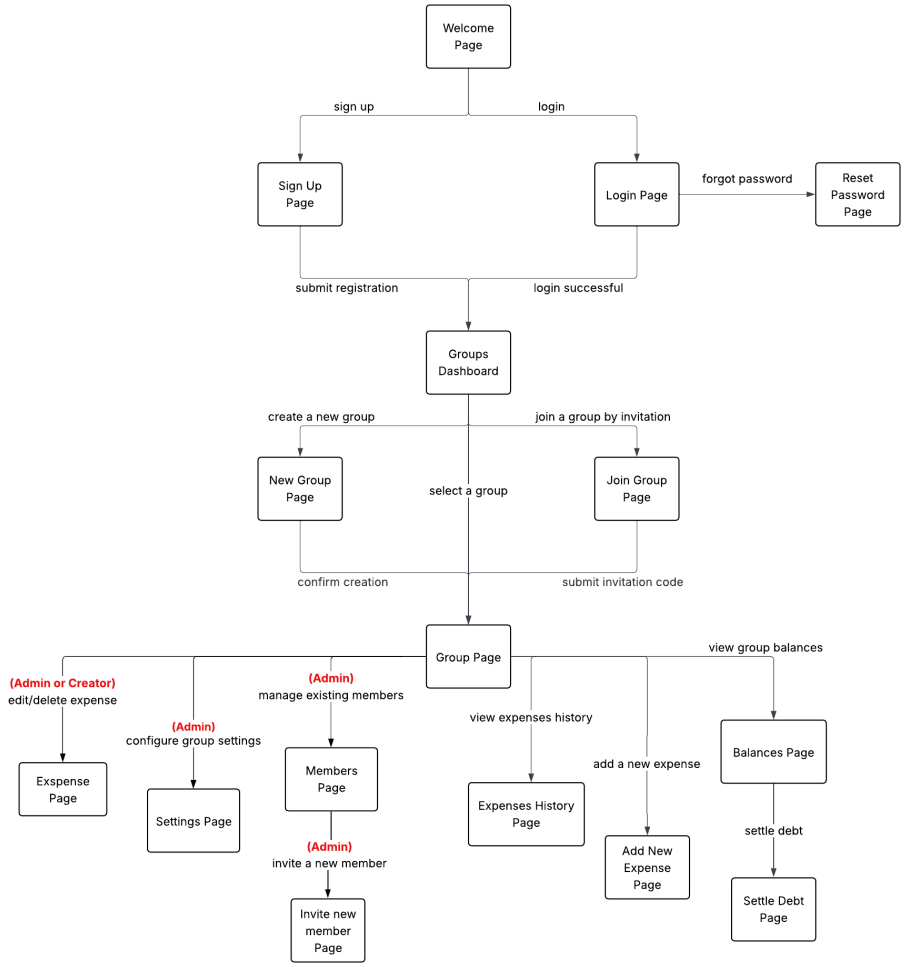

Figura 9: Page Navigation Diagram

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

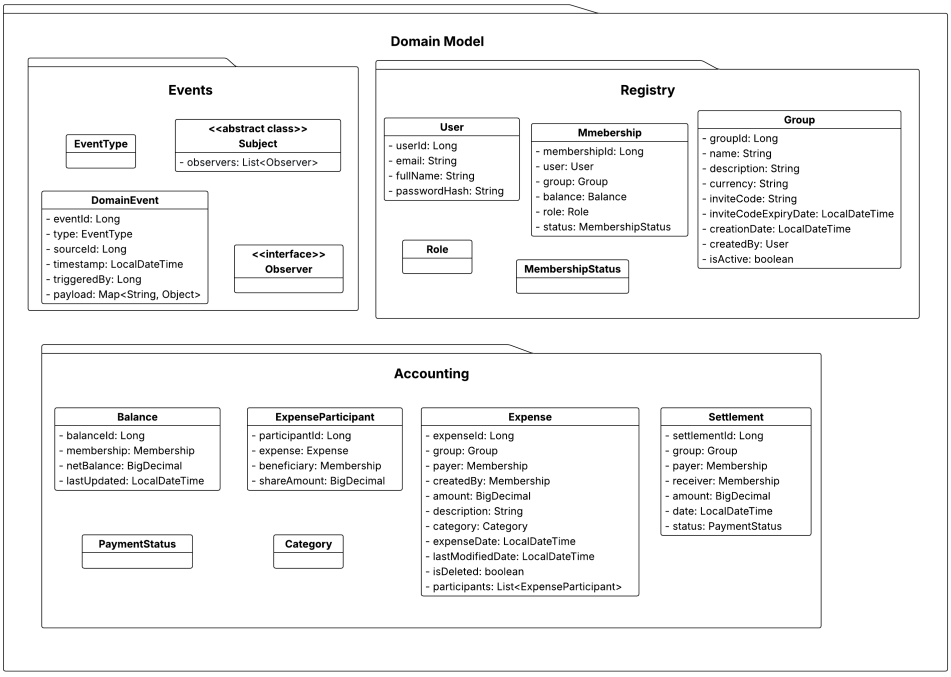

Figura 10: Suddivisione del Domain Model in sotto-package

- **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.

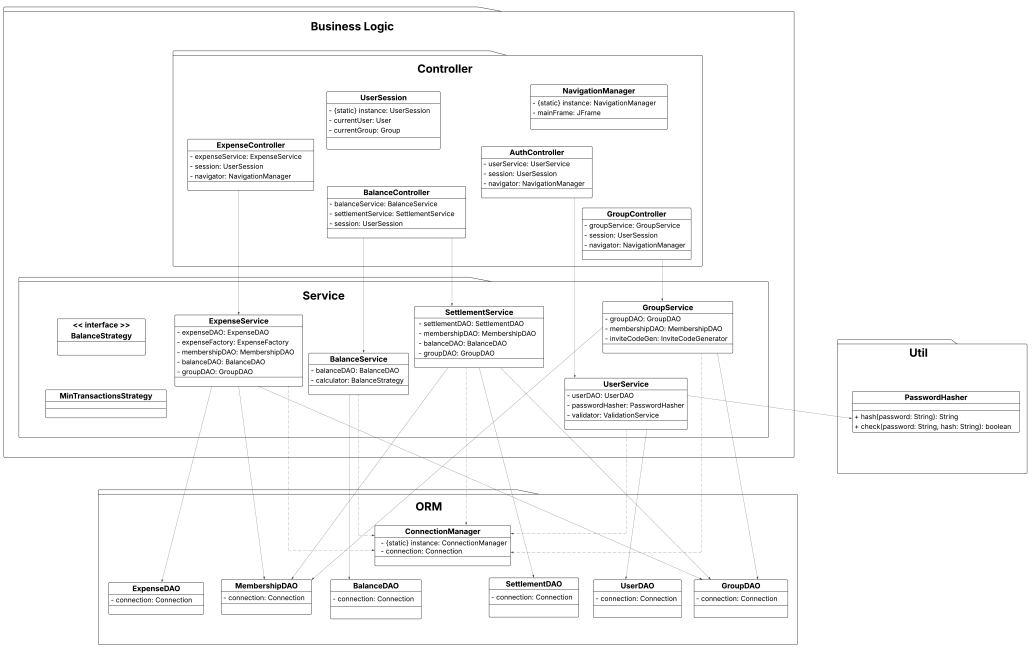

Figura 11: Dipendenze tra i package Controller, Service, ORM e Util

- **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

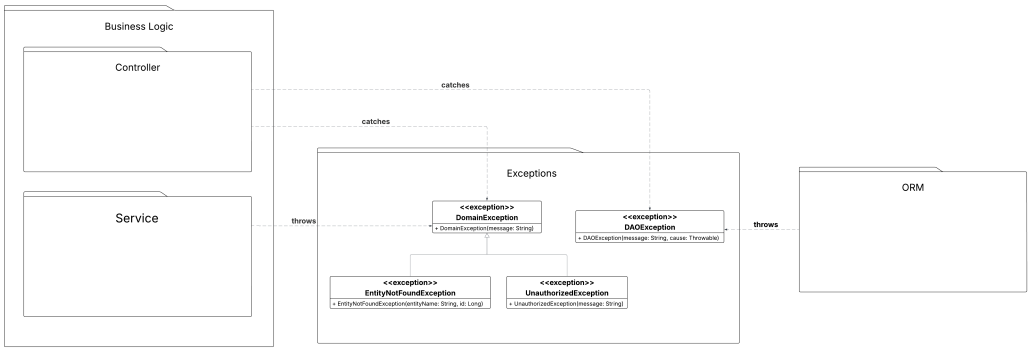

Figura 12: Exceptions Package

non dipende dagli altri layer. Le eccezioni sono condivise tra i layer per rappresentare condizioni di errore applicativo.

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).

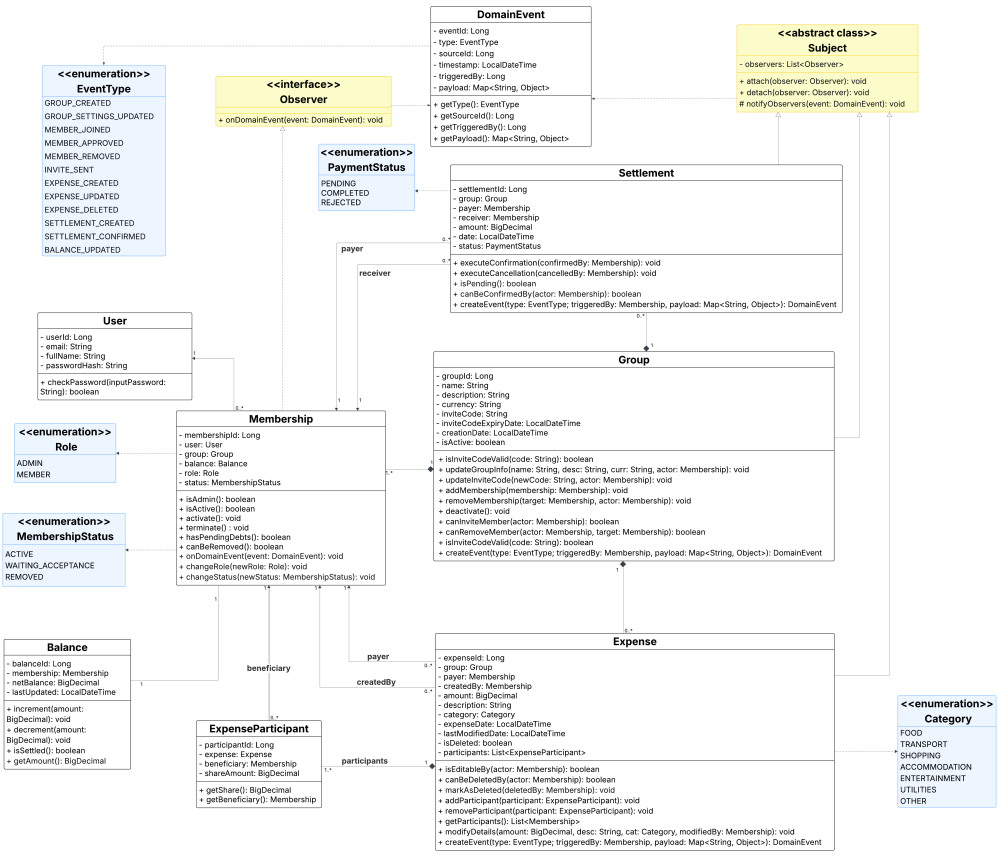

Figura 13: Domain Model Class Diagram

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

- **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.

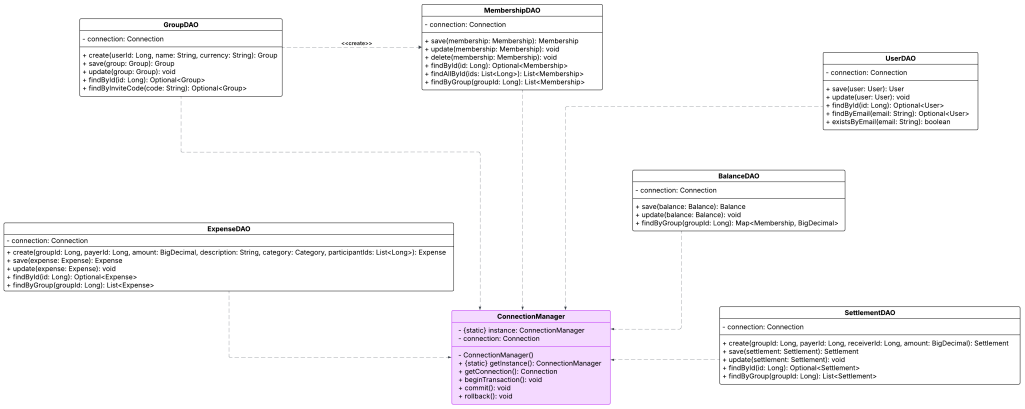

Figura 14: ORM Class Diagram

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 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 (Figura 14), i DAO includono metodi specifici per il dominio, quali `findByInviteCo`

de 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 (`R esultSet`) in oggetti Java utilizzabili dai layer superiori.

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, al Service Layer sono assegnate due responsabilità architetturali fondamentali:

1. Gestione transazionale: I Service definiscono i confini delle transazioni atomiche interagendo con il `ConnectionManager`, garantiscono che operazioni che coinvolgono scritture multiple su diversi DAO vengano confermate (commit) o annullate (rollback) in blocco, preservando l'integrità del database.

2. Ciclo di vita degli Observer: 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 caricando i membri interessati e agganciandoli al Subject.

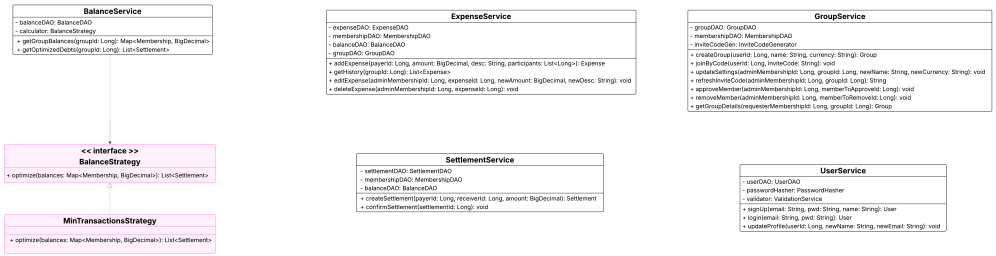

Figura 15: Service Layer Class Diagram

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 rispetta il principio **Open/Closed**: qualora in futuro si rendesse necessario introdurre un nuovo criterio per il calcolo dei rimborsi

(ad esempio, un algoritmo che arrotonda le cifre per agevolare lo scambio di contanti), sarà sufficiente sviluppare una nuova classe che implementi `BalanceStrategy`, lasciando invariato il codice del `BalanceService`.

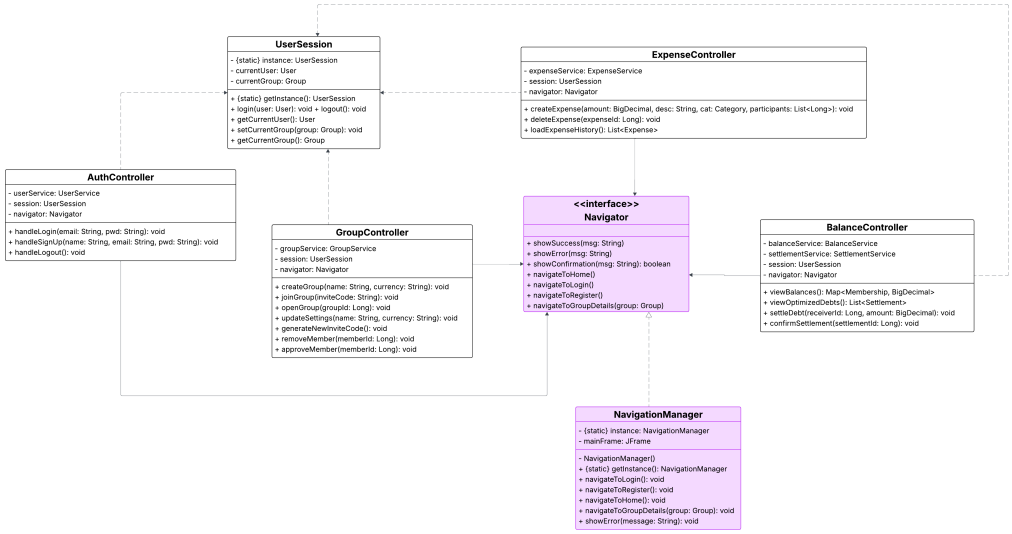

Figura 16: Controller Layer Class Diagram

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

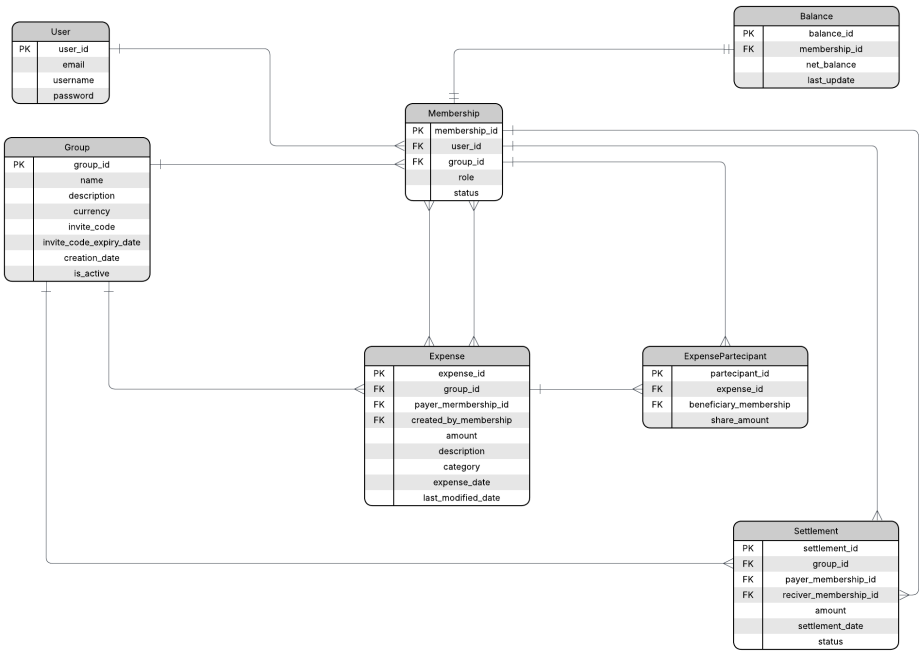

Figura 17: Modello ER

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 architetturali definite in fase di progettazione, con l'obiettivo di mantenere una chiara separazione delle responsabilità tra i diversi livelli applicativi.

4.1 Domain Model

Il Domain Model è stato implementato in Java ponendo particolare attenzione alla protezione delle invarianti di classe, all'uso corretto dei tipi per i dati monetari e all'integrazione con il pattern Observer tramite una lista di observer non persistita.

4.1.1 Protezione delle invarianti nei costruttori

Ogni entità del dominio valida i propri invarianti direttamente nel costruttore, rendendo impossibile la creazione di oggetti in uno stato inconsistente. La filosofia adottata è quella del *fail-fast*: se i dati forniti violano le regole di business, viene lanciata immediatamente un'eccezione prima che l'oggetto venga allocato.

La classe `User` verifica che email, nome e hash della password non siano nulli o vuoti tramite controlli espliciti; `Membership` richiede che `user`, `group` e `role` siano tutti non-null, sempre tramite controlli espliciti; `Group` lancia `IllegalArgumentException` se nome o valuta sono assenti. L'invariante più critica si trova in `Balance`: il campo `membership` è dichiarato `fina l` e protetto con `Objects.requireNonNull()`, rendendo strutturalmente impossibile avere un saldo orfano. Il campo `netBalance` viene inizializzato a zero con scala fissa a due cifre decimali, garantendo la consistenza monetaria fin dalla costruzione:

```
1 public Balance(Long balanceId, Membership membership) {
2     this.balanceId = balanceId;
3     this.membership = Objects.requireNonNull(membership);
4     this.netBalance = BigDecimal.ZERO.setScale(2,
    RoundingMode.HALF_UP);
5     this.lastUpdated = LocalDateTime.now();
6 }
```

Listing 1: Invarianti del costruttore di Balance

In `Expense`, la validazione dell'importo è incapsulata nel metodo privato `setAmount()`, che viene invocato sia nel costruttore sia in `modifyDetails( )`: in questo modo la regola è espressa in un unico punto e non può essere

aggirata nemmeno durante un aggiornamento successivo. Analogamente, `Settlement` verifica nel costruttore che `payer` e `receiver` siano membri distinti, proteggendo il sistema da auto-pagamenti:

```
// In Expense: metodo privato richiamato nel costruttore e in modifyDetails()
private void setAmount(BigDecimal amount) {
    if (amount == null || amount.compareTo(BigDecimal.ZERO) <= 0) {
        throw new DomainException("Amount must be positive.");
    }
    this.amount = amount;
}

// In Settlement: controllo di identita' nel costruttore
if (payer.equals(receiver)) {
    throw new IllegalArgumentException("Payer and receiver must be different");
}
```

Listing 2: Validazione singola di importo in Expense e verifica di identita in Settlement

4.1.2 Lista di Observer non persistita e wiring a runtime

La classe astratta `Subject` mantiene la lista degli Observer come campo d'istanza ordinario, senza alcuna marcatura speciale nel codice Java. L'assenza di persistenza di questo campo è una conseguenza architettonica deliberata: il Data Access Layer ricostruisce le entità a partire dai dati relazionali, e nessuna query SQL carica relazioni di tipo Observer. La lista è pertanto *logicamente transient*, una caratteristica architetturale che prescinde dall'uso della keyword `transient` di Java, la quale non è necessaria in assenza di serializzazione.

La conseguenza pratica è che ogni entità `Subject` (`Group`, `Expense`, `Settlement`) viene restituita dal DAO con la lista degli observer vuota. Il Service Layer si fa carico di ricollegare gli observer prima di invocare qualsiasi operazione che generi eventi, attraverso la procedura di wiring descritta nella Sezione 4.3.2.

La classe `Subject` adotta inoltre una misura difensiva nella notifica: `notifyObservers()` opera su una copia della lista per evitare eccezioni di tipo `ConcurrentModificationException` nel caso in cui un observer modificasse la lista durante la propagazione dell'evento:

```java
protected void notifyObservers(DomainEvent event) {
    if (event == null) return;
    List<Observer> observersCopy = new ArrayList<>(observers)
    ;
    for (Observer observer : observersCopy) {
        try {
            observer.onDomainEvent(event);
        } catch (Exception e) {
            // Continua con gli altri observer
        }
    }
}
```

Listing 3: Notifica sicura degli observer in Subject

4.1.3 Soft delete in Expense

La cancellazione delle spese è implementata come *soft delete*: il metodo `markAsDeleted(MembershipdeletedBy)` imposta il flag booleano `isDeleted=true` sull'oggetto e aggiorna `lastModifiedDate`, senza rimuovere alcuna riga dal database. Questo approccio preserva la storia contabile del gruppo e consente di tracciare chi ha eliminato la spesa e quando.

Prima di procedere, il metodo verifica l'autorizzazione tramite `canBeDeletedBy()`: solo l'admin del gruppo o chi ha creato la spesa può eliminarla. Se la spesa è già stata eliminata, viene sollevata una `IllegalStateException` per evitare cancellazioni doppie. Successivamente viene generato e propagato un `DomainEvent` di tipo `EXPENSE_DELETED` agli Observer collegati; la notifica raggiunge le `Membership` agganciate, che la utilizzano per simulare l'invio di una comunicazione all'utente tramite logging. Le rettifiche contabili sui `Balance` sono invece gestite transazionalmente dal Service Layer, che dopo la notifica aggiorna i saldi tramite `BalanceDAO`.

```java
public void markAsDeleted(Membership deletedBy) {
    if (!canBeDeletedBy(deletedBy)) {
        throw new UnauthorizedException(
            "Only admin or creator can delete the expense");
    }
    if (isDeleted) {
        throw new IllegalStateException("Expense already
    deleted");
    }
    this.isDeleted = true;
    touch(); // aggiorna lastModifiedDate
    notifyObservers(createEvent(
        EventType.EXPENSE_DELETED,
```

```
13          deletedBy,
14          Map.of("amount", amount, "description", description)
15      ));
16  }
```

Listing 4: Soft delete in Expense

A livello di persistenza, il metodo `findByGroup()` in `ExpenseDAO` filtra automaticamente le spese eliminate attraverso la clausola `WHEREis_deleted=FALSE`, rendendo la cancellazione logica trasparente per i layer superiori senza alcun intervento aggiuntivo del Service Layer.

4.1.4 Uso di BigDecimal per i calcoli monetari

Tutti i campi e le operazioni che coinvolgono importi monetari utilizzano `java.math.BigDecimal` invece di `double` o `float`. La scelta è motivata dalla necessità di evitare gli errori di rappresentazione in virgola mobile tipici dei tipi primitivi: una somma come `0.1+0.2` in `double` produce `0.30000000000000004`, un risultato inaccettabile in contesti finanziari.

`BigDecimal` garantisce precisione decimale arbitraria e operazioni di arrotondamento configurabili. Nel sistema, il modo di arrotondamento standard adottato è `RoundingMode.HALF_UP` con scala fissa a 2 cifre decimali. Questo arrotondamento è applicato sistematicamente nei metodi `increment()`, `decrement()` e `apply()` di `Balance`, come mostra il seguente estratto:

```
1  public void increment(BigDecimal amount) {
2      validateAmount(amount);
3      netBalance = netBalance.add(amount).setScale(2,
       RoundingMode.HALF_UP);
4      touch();
5  }
6
7  public void decrement(BigDecimal amount) {
8      validateAmount(amount);
9      netBalance = netBalance.subtract(amount).setScale(2,
       RoundingMode.HALF_UP);
10     touch();
11 }
```

Listing 5: Arrotondamento sistematico in Balance.increment() e decrement()

Il medesimo criterio si applica nel calcolo della quota per partecipante in `ExpenseDAO.create()`, dove si usa `BigDecimal.divide()` con scala e `RoundingMode` espliciti, e nella classe `MinTransactionsStrategy`, dove il confronto tra importi viene effettuato sempre con `compareTo()` invece dell'operatore `==`, rispettando la semantica di `BigDecimal`.

4.2 Data Access Layer

Il Data Access Layer gestisce la persistenza del sistema attraverso il pattern DAO (Data Access Object). Ogni classe DAO incapsula le query SQL relative a una specifica entità di dominio, esponendo al Service Layer un'interfaccia orientata agli oggetti che nasconde completamente i dettagli del database relazionale sottostante.

4.2.1 ConnectionManager: Singleton e gestione della connessione JDBC

La classe `ConnectionManager` è implementata come Singleton con inizializzazione *eager*: l'istanza unica viene creata al momento del caricamento della classe tramite un campo `staticfinal`, garantendo thread-safety senza necessità di sincronizzazione esplicita.

```
1  public class ConnectionManager {
2
3      private static final ConnectionManager INSTANCE = new
       ConnectionManager();
4      private Connection connection;
5
6      private ConnectionManager() {
7          try {
8              connection = DriverManager.getConnection(
9                  "jdbc:h2:mem:splitmanager;DB_CLOSE_DELAY=-1;"
10                 + "INIT=RUNSCRIPT FROM 'classpath:schema.sql'
       ",
11                 "sa", ""
12             );
13             connection.setAutoCommit(false);
14         } catch (SQLException e) {
15             throw new RuntimeException("Failed to connect to
       database", e);
16         }
17     }
18
19     public Connection getConnection() { return connection; }
20
21     public static ConnectionManager getInstance() { return
       INSTANCE; }
22
23     public void beginTransaction() throws SQLException {
24         connection.setAutoCommit(false);
25     }
26     public void commit() throws SQLException {
27         connection.commit();
```

```
28          connection.setAutoCommit(true);
29      }
30      public void rollback() throws SQLException {
31          connection.rollback();
32          connection.setAutoCommit(true);
33      }
34  }
```

Listing 6: Singleton con inizializzazione eager in ConnectionManager

La connessione è diretta a un database H2 in-memory, scelto per semplicità di setup e velocità di esecuzione nei test. Il parametro `DB_CLOSE_DELAY=-1` mantiene il database attivo per tutta la durata del processo JVM. L'autocommit è disabilitato fin dal costruttore, cosicché ogni operazione si colloca di default in una transazione controllata dal Service Layer. I metodi `beginTransaction()`, `commit()` e `rollback()` permettono al Service di gestire i confini transazionali senza accedere direttamente all'oggetto `Connection`.

Tutti i DAO ottengono la connessione chiamando `getInstance().getConnection()` sul Singleton `ConnectionManager`, garantendo che l'intera applicazione condivida un unico oggetto `Connection` e che tutte le operazioni eseguite nell'ambito di un caso d'uso siano parte della stessa transazione.

4.2.2 Struttura comune dei DAO e mapping ResultSet – entità

Ogni DAO segue una struttura uniforme composta da tre responsabilità principali: le operazioni CRUD, la costruzione delle query SQL tramite `PreparedStatement`, e il mapping dei risultati da `ResultSet` a oggetti di dominio.

Il mapping è centralizzato in un metodo privato `mapResultSetToX()` per ciascun DAO, che ricostruisce l'oggetto di dominio leggendo le colonne del `ResultSet` per nome. Questo approccio isola il codice di conversione in un unico punto, facilitando la manutenzione in caso di modifiche allo schema.

Per le entità con relazioni, come `Membership` che aggrega `User` e `Group`, il DAO utilizza JOIN nella query SQL ed esegue il mapping annidato all'interno dello stesso metodo, ricostruendo l'oggetto completo in un'unica passata sul `ResultSet`. Il metodo `mapResultSetToMembership()` in `MembershipDAO` illustra la complessità tipica di questo mapping: costruisce prima un oggetto `User` e un oggetto `Group` a partire dalle colonne della JOIN, gestisce la conversione del campo nullable `invite_code_expiry_date` da `Timestamp` a `LocalDateTime`, converte il ruolo da stringa a enum tramite `Role.valueOf()`, e infine assembla la `Membership` impostando lo stato corretto:

```
1  private Membership mapResultSetToMembership(ResultSet rs)
       throws SQLException {
```

```
2       User user = new User(
3           rs.getLong("user_id"),
4           rs.getString("email"),
5           rs.getString("full_name"),
6           rs.getString("password_hash")
7       );
8
9       Group group = new Group(
10          rs.getLong("group_id"),
11          rs.getString("name"),
12          rs.getString("currency")
13      );
14      group.setDescription(rs.getString("description"));
15      group.setInviteCode(rs.getString("invite_code"));
16
17      // Conversione nullable Timestamp -> LocalDateTime
18      Timestamp expiry = rs.getTimestamp("
        invite_code_expiry_date");
19      if (expiry != null) {
20          group.setInviteCodeExpiry(expiry.toLocalDateTime());
21      }
22      group.setActive(rs.getBoolean("is_active"));
23
24      Membership membership = new Membership(
25          rs.getLong("membership_id"),
26          user,
27          group,
28          Role.valueOf(rs.getString("role"))
29      );
30
31      MembershipStatus status = MembershipStatus.valueOf(rs.
        getString("status"));
32      membership.changeStatus(status);
33
34      return membership;
35  }
```

Listing 7: Mapping annidato in MembershipDAO.mapResultSetToMembership()

Le operazioni di inserimento recuperano la chiave primaria autogenerata dal database tramite `Statement.RETURN_GENERATED_KEYS` e `stmt.getGeneratedKeys()`, assegnandola all'oggetto di dominio prima di restituirlo al chiamante. In questo modo il Service Layer riceve sempre oggetti completamente identificati senza dover eseguire una query aggiuntiva.

Infine, il `BalanceDAO` adotta un pattern di *self-contained mapping*: invece di dipendere da `MembershipDAO` per ricostruire le `Membership` associate ai saldi (il che creerebbe una dipendenza circolare), replica localmente il codice

di mapping necessario. La duplicazione è consapevole e documentata, ed è preferita rispetto all'introduzione di un ciclo di dipendenze tra DAO.

4.2.3 Gestione SQLException e traduzione in DAOException

Ogni metodo DAO avvolge le proprie operazioni SQL in un blocco `try-catch` che intercetta le eccezioni `SQLException` di basso livello e le converte in `DAOException`, un'eccezione unchecked definita nel package `exception`. Questa traduzione ha una funzione architetturale precisa: isola il Service Layer dai dettagli dell'infrastruttura di persistenza, consentendogli di gestire gli errori a un livello di astrazione più alto senza dover conoscere i codici di errore JDBC o le eccezioni checked di `java.sql`. Inoltre, la `SQLException` originale viene passata al costruttore della `DAOException` come causa, preservando così la traccia dello stack dell'errore originario.

Il metodo `save()` di `UserDAO` illustra il pattern: la chiamata a `stmt.executeUpdate()` è racchiusa in un `try-with-resources` che chiude automaticamente il `PreparedStatement` al termine, mentre l'eventuale `SQLException` viene catturata e rilasciata come `DAOException` con un messaggio descrittivo e la causa originale incapsulata nel campo `cause`:

```
1  public User save(User user) {
2      String sql = "INSERT INTO users (email, full_name,
       password_hash) "
3                  + "VALUES (?, ?, ?)";
4      try (PreparedStatement stmt = connection.prepareStatement
       (
5              sql, Statement.RETURN_GENERATED_KEYS)) {
6
7          stmt.setString(1, user.getEmail());
8          stmt.setString(2, user.getFullName());
9          stmt.setString(3, user.getPasswordHash());
10
11         int affectedRows = stmt.executeUpdate();
12         if (affectedRows == 0) {
13             throw new DAOException("Creating user failed",
       null);
14         }
15
16         try (ResultSet keys = stmt.getGeneratedKeys()) {
17             if (keys.next()) {
18                 Long userId = keys.getLong(1);
19                 return new User(userId, user.getEmail(),
20                                 user.getFullName(), user.
       getPasswordHash());
21             } else {
```

```
22                  throw new DAOException("No ID obtained for
    new user", null);
23              }
24          }
25      } catch (SQLException e) {
26          throw new DAOException("Error saving user", e);
27      }
28  }
```

Listing 8: Traduzione di SQLException in DAOException in UserDAO.save()

Oltre alla semplice conversione, alcuni DAO implementano logiche di persistenza più avanzate. Il `BalanceDAO`, ad esempio, gestisce un caso particolare: il metodo `save()` tenta prima un `INSERT` e, se riceve un errore di constraint violation (indicato dallo SQL state `23505` per H2 o dall'error code `1062` per MySQL), interpreta l'eccezione e si comporta come un *upsert*, delegando automaticamente l'operazione a `update()`.

```
1   public Balance save(Balance balance) {
2       String sql = "INSERT INTO balances (membership_id,
    net_balance, last_updated) VALUES (?, ?, ?)";
3       try (PreparedStatement stmt = connection.prepareStatement
    (sql, Statement.RETURN_GENERATED_KEYS)) {
4           stmt.setLong(1, balance.getMembership().
    getMembershipId());
5           stmt.setBigDecimal(2, balance.getNetBalance());
6           stmt.setTimestamp(3, Timestamp.valueOf(LocalDateTime.
    now()));
7
8           int affectedRows = stmt.executeUpdate();
9           if (affectedRows == 0) {
10              throw new DAOException("Creating balance failed",
     null);
11          }
12
13          try (ResultSet keys = stmt.getGeneratedKeys()) {
14              if (keys.next()) {
15                  Long balanceId = keys.getLong(1);
16                  balance.setBalanceId(balanceId);
17              }
18          }
19
20          return balance;
21      } catch (SQLException e) {
22          if (e.getErrorCode() == 1062 || "23505".equals(e.
    getSQLState())) {
23              return update(balance);
24          }
```

```
25            throw new DAOException("Error saving balance", e);
26        }
27    }
```

Listing 9: Logica di upsert in BalanceDAO.save()

Questo meccanismo semplifica la logica del Service Layer, che può invocare `balanceDAO.save()` senza preoccuparsi se il record esista già o meno, delegando al DAO la responsabilità di eseguire l'operazione corretta.

4.3 Service Layer

Il Service Layer funge da collante tra la logica di business pura (Domain) e l'accesso ai dati (DAO). Di seguito si analizzano i dettagli implementativi più rilevanti di questo livello.

4.3.1 Gestione Delle Transazioni

Alla base della gestione transazionale c'è la classe Singleton `ConnectionManager` del livello DAO, che gestisce l'unica connessione attiva verso il database H2 mettendo a disposizione i metodi `beginTransaction()`, `commit()` e `rollback()`.

La responsabilità di invocare questi metodi spetta alle classi Service che, essendo a conoscenza dell'intero caso d'uso, decidono i confini della transazione coordinando le chiamate a più DAO. Per farlo in modo sicuro, il Service sfrutta l'istanza del `ConnectionManager` all'interno di un costrutto `try-catch`, orchestrando così il `commit` in caso di successo o il `rollback` in caso di errore.

Il flusso standard di una transazione gestita dal Service funziona in questo modo:

1. Il Service ottiene la connessione dal `ConnectionManager` e invoca `beginTransaction()` prima di eseguire qualsiasi operazione che modifichi lo stato del database.

2. Il Service chiama i vari DAO necessari per compiere l'azione richiesta dall'utente.

3. Se tutte le operazioni sui DAO vanno a buon fine, il Service invoca `commit()` per rendere permanenti le modifiche.

4. Se si verifica un'eccezione durante l'esecuzione delle operazioni sui DAO (ad esempio per errori SQL o vincoli violati), il Service cattura l'eccezione, invoca `rollback()` per annullare tutte le modifiche parziali e

propaga l'errore incapsulandolo in una `DAOException`. Questo garantisce l'integrità dei dati e isola i livelli superiori dai dettagli infrastrutturali del database.

Nel seguente snippet, tratto da `ExpenseService`, è visibile la gestione transazionale durante la creazione di una nuova spesa:

```
ConnectionManager connMgr = ConnectionManager.getInstance();

try {
    connMgr.beginTransaction(); // Disabilita l'autocommit

    // [Omissis: validazione e caricamento entita']

    Expense expense = expenseDAO.create(
        groupId, payerMembershipId, amount,
        description, category, participantIds
    );

    // [Omissis: Wiring degli Observer e chiamate a
    balanceDAO.update()]

    connMgr.commit(); // Conferma le modifiche in blocco
    return expense;

} catch (Exception e) {
    try {
        connMgr.rollback(); // Annulla tutte le modifiche
    parziali
    } catch (SQLException ex) {
        throw new DAOException("Error during transaction
    rollback", ex);
    }

    if (e instanceof DomainException) {
        throw (DomainException) e;
    }
    throw new DomainException("Error creating expense", e);
}
```

Listing 10: Gestione della transazione nel metodo addExpense di ExpenseService

4.3.2 Wiring Dinamico degli Observer

Il pattern Observer viene utilizzato per fare in modo che determinati eventi aggiornino in automatico altre entità. Le classi Group, Expense e Settlement

fungono da **Subject**, mentre la classe Membership agisce come **Observer**.

All'interno della classe astratta `Subject`, la lista degli Observer non viene persistita nel database (come discusso nella Sezione 4.1.2). Di conseguenza, quando un'entità come una `Expense` viene letta dal database, non sa chi siano i suoi Observer, è responsabilità esclusiva del Service Layer ricreare questi collegamenti al volo (operazione definita appunto "wiring"). Dunque, quando deve eseguire un'operazione che scaturirà un evento (es. creazione di una nuova spesa), il Service esegue questi passaggi precisi:

1. Interroga il database tramite i DAO per ottenere l'oggetto principale (ad esempio l'Expense).
2. Carica tutte le entità Observer correlate (tutte le Membership coinvolte).
3. Effettua il wiring, ovvero collega manualmente gli oggetti in memoria chiamando il metodo `attach()` sul Subject per ogni Observer trovato (es. textttexpense.attach(membership)).
4. Esegue la modifica richiesta. A questo punto, il Subject invoca internamente `notifyObservers()`; le Membership collegate ricevono la notifica in tempo reale e aggiornano i propri saldi.
5. Una volta che gli Observer hanno aggiornato il loro stato in memoria, il Service utilizza i DAO appropriati (es. BalanceDAO) per salvare le modifiche definitive sul database.

```
1 // Carica tutti i membri del gruppo dal database
2 List<Membership> allMembers = membershipDAO.findByGroup(
    groupId);
3
4 for (Membership member : allMembers) {
5     expense.attach(member); // Effettua il Wiring
6 }
```

Listing 11: Implementazione del Wiring dinamico in ExpenseService

Gestione della memoria (Request Scope) Poiché il wiring viene effettuato dinamicamente, gli oggetti coinvolti hanno un ciclo di vita limitato alla singola operazione applicativa (*Request Scope*). Il Service li carica dal database, li collega tramite `attach()`, esegue l'operazione richiesta e non mantiene riferimenti persistenti oltre tale ambito, consentendo al sistema di liberarli automaticamente al termine dell'elaborazione, riducendo così il rischio di memory leak.

4.3.3 Ottimizzazione dei Debiti: Algoritmo Greedy

L'algoritmo di ottimizzazione incapsulato in `MinTransactionsStrategy` implementa un approccio **greedy**.

Il metodo `optimize()` riceve in ingresso la mappa dei saldi netti (`Map<Membership,BigDecimal>`) e suddivide i membri in due insiemi: **Debitors** (saldo negativo) e **Creditors** (saldo positivo), utilizzando una classe di supporto interna chiamata `DebtorCredit`. Le due liste vengono ordinate in ordine decrescente rispetto all'importo. Successivamente, un ciclo `while` effettua un matching greedy tra il debitore con debito maggiore e il creditore con credito maggiore, creando un nuovo oggetto `Settlement` per l'importo minimo tra i due saldi e aggiornando i residui fino all'azzeramento completo.

```
// Matching greedy (su liste precedentemente ordinate)
int i = 0; // Indice debitori
int j = 0; // Indice creditori

while (i < debtors.size() && j < creditors.size()) {
    DebtorCredit debtor = debtors.get(i);
    DebtorCredit creditor = creditors.get(j);

    // L'importo da trasferire e' il minimo tra debito e credito attuale
    BigDecimal settleAmount = debtor.amount.min(creditor.amount);

    // Registra la transazione da effettuare
    settlements.add(new Settlement(
        debtor.membership.getGroup(), debtor.membership, creditor.membership, settleAmount
    ));

    // Decurta l'importo appena saldato dai totali residui
    debtor.amount = debtor.amount.subtract(settleAmount);
    creditor.amount = creditor.amount.subtract(settleAmount);

    // Se il debito o il credito e' azzerato, si avanza con l'indice
    if (debtor.amount.compareTo(BigDecimal.ZERO) == 0) i++;
    if (creditor.amount.compareTo(BigDecimal.ZERO) == 0) j++;
}
```

Listing 12: Nucleo del matching Greedy in MinTransactionsStrategy

Questa implementazione garantisce matematicamente che il numero di transazioni generate sia sempre al più $(N - 1)$, dove (N) è il numero di membri coinvolti

4.4 Controller Layer e CLI

Il Controller Layer costituisce il punto di ingresso della logica applicativa, ricevendo gli input dall'interfaccia utente e coordinando le chiamate ai Service. Il sistema utilizza una **Command Line Interface (CLI)** come Presentation Layer.

L'architettura adottata separa nettamente questi due livelli, garantendo che:

- I Controller contengano esclusivamente logica di coordinamento, delegando ai Service le operazioni di business.
- La CLI gestisca solo aspetti di presentazione (menu, input/output), senza conoscere dettagli implementativi dei Service o del Domain.

4.4.1 Dependency Injection nei Controller

Tutti i Controller ricevono le proprie dipendenze (Service, Session, Navigator) tramite costruttore, applicando il principio di **Dependency Injection**. Questo approccio offre numerosi vantaggi in termini di testabilità ed esplicitezza, garantendo inoltre l'immutabilità dello stato (le dipendenze sono dichiarate `final`):

```
1  public class AuthController {
2      private final UserService userService;
3      private final UserSession session;
4      private final Navigator navigator;
5
6      public AuthController(UserService userService,
7                            UserSession session,
8                            Navigator navigator) {
9          if (userService == null || session == null ||
       navigator == null) {
10             throw new IllegalArgumentException("Dependencies
       cannot be null");
11         }
12         this.userService = userService;
13         this.session = session;
14         this.navigator = navigator;
15     }
16     // ... logica di coordinamento ...
17 }
```

Listing 13: Dependency Injection nel costruttore di AuthController

Nel codice sopra, `AuthController` riceve tre dipendenze tramite costruttore:

- `UserService`: per eseguire operazioni di autenticazione.
- `UserSession`: per mantenere lo stato della sessione corrente.
- `Navigator`: per gestire navigazione e feedback utente.

Il costruttore valida che nessuna dipendenza sia `null`, garantendo che il Controller sia sempre in uno stato consistente.

4.4.2 Gestione dello Stato: UserSession Singleton

Il mantenimento dello stato dell'utente autenticato è affidato alla classe `UserSession`, implementata seguendo il pattern creazionale **Singleton**. Questa specifica architettura prevede un costruttore privato, che impedisce la creazione diretta di istanze esterne tramite la keyword `new`.

L'istanza unica viene invece allocata in memoria solo al primo accesso effettivo tramite il metodo `getInstance()`, sfruttando il meccanismo della *lazy initialization* per ottimizzare il consumo di risorse. Tale approccio garantisce l'esistenza di un'unica istanza globale condivisa per l'intera esecuzione del programma: ciò permette ai vari Controller di accedere in qualsiasi momento ai dati dell'utente loggato e del gruppo correntemente selezionato, eliminando la necessità di propagare continuamente questi oggetti attraverso le firme dei metodi.

Infine, la classe espone metodi di utility dedicati, come `isLoggedIn()` e `hasGroupSelected()`, che semplificano notevolmente i controlli condizionali all'interno dei layer superiori.

```
1  public class UserSession {
2      private static UserSession instance;
3      private User currentUser;
4      // ... altri campi ...
5
6      private UserSession() {} // Costruttore privato
7
8      public static UserSession getInstance() {
9          if (instance == null) {
10             instance = new UserSession(); // Istanziazione
       ritardata (Lazy)
11         }
12         return instance;
13     }
14     // ... getter e setter ...
15 }
```

Listing 14: Lazy initialization nel Singleton UserSession

4.4.3 Dependency Inversion Principle (DIP)

Per mantenere la logica applicativa isolata dal layer di input/output su console, il design applica rigorosamente il **Dependency Inversion Principle (DIP)**.

Anziché far dipendere i Controller dall'implementazione concreta che gestisce i menu, li si fa dipendere dall'interfaccia astratta `Navigator`. In produzione, l'entry-point dell'applicazione inietta nei Controller il `Navigatio nManager` standard, che gestisce concretamente il flusso della CLI e le stampe a schermo.

Oltre a favorire la modularità, questa scelta architetturale è stata pensata primariamente per abilitare la testabilità automatizzata del sistema. Disaccoppiando l'I/O, è infatti possibile iniettare nei Controller un componente fittizio (`StubNavigator`) che sopprime l'output e non blocca il thread in attesa di input da tastiera. L'implementazione e l'utilizzo pratico di questo Stub durante i test End-to-End sono approfonditi nella Sezione 5.3.2.

4.4.4 Architettura della Presentation (CLI)

La modularità e la robustezza del livello di presentazione (CLI) sono garantite da due scelte implementative chiave che azzerano la duplicazione del codice:

- **InputHandler centralizzato**: Una classe di utilità che incapsula l'uso di `java.util.Scanner`. Si occupa di validare superficialmente gli input (controllando formati numerici, stringhe vuote, ecc.) e implementa logiche di *retry* automatico in caso di inserimento errato. In questo modo i Controller ricevono sempre dati "puliti" e correttamente tipizzati.
- **Struttura Gerarchica dei Menu**: Il flusso applicativo è orchestrato da un coordinatore principale (`CLIMenu`) che agisce da router verso sottomenu specializzati (`AuthMenu`, `GroupMenu`, `ExpenseMenu`, ecc.). Ciascun sottomenu riceve l'`InputHandler` e le istanze dei Controller necessari, garantendo alta coesione e rispetto del principio di singola responsabilità (SRP).

```
1 public BigDecimal readAmount(String prompt) {
2     while (true) { // Loop di retry
3         try {
4             System.out.print(prompt);
5             String input = scanner.nextLine().trim();
6             BigDecimal amount = new BigDecimal(input);
7 
8             if (amount.compareTo(BigDecimal.ZERO) > 0) {
9                 return amount; // Input valido, esce dal loop
```

```
10              }
11              System.out.println("[ERROR] L'importo deve essere
     positivo.");
12          } catch (NumberFormatException e) {
13              System.out.println("[ERROR] Formato non valido.
    Inserisci un numero.");
14          }
15      }
16  }
```

Listing 15: Esempio di logica di retry automatico nell'InputHandler

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".

```
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
    ());
}
```

Listing 16: Test della logica di pareggio del saldo

- `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`).

```
void testConstructor_WithNegativeAmount_ThrowsException()
    {
    assertThrows(
        DomainException.class,
        () -> new Expense(
            1L, group, creator, creator,
            new BigDecimal("-100.00"),
```

```
7               "Invalid Expense",
8               Category.FOOD,
9               LocalDateTime.now()
10          )
11      );
12  }
13
```

Listing 17: Test del costruttore con importo negativo

- `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.

```
1  void canBeRemoved_reflectsPendingDebts() {
2      Membership m = new Membership(null, userOther, group,
       Role.MEMBER);
3      assertTrue(m.canBeRemoved());
4
5      Balance b = new Balance(null, m);
6      b.increment(new BigDecimal("1.00"));
7      m.setBalance(b);
8
9      assertFalse(m.canBeRemoved());
10 }
```

Listing 18: Test della logica di rimozione di una Membership con debiti pendenti

5.2 Test di Integrazione (Grey-Box)

I test di integrazione verificano il corretto funzionamento dell'interazione tra **Service Layer**, **DAO Layer** e **Database reale**, testando Use Case completi con componenti autentici, senza l'ausilio di mock per il livello di persistenza.

5.2.1 BaseIntegrationTest: Classe Astratta

Tutti i test di integrazione estendono BaseIntegrationTest. Questa classe si occupa di inizializzare la connessione al database H2 in-memory, istanziare tutti i DAO reali e pulire l'intero schema relazionale prima di ogni test tramite `@BeforeEach`

```java
@BeforeEach
void setUp() throws Exception {
    connection = ConnectionManager.getInstance().
    getConnection();
    userDAO = new UserDAO();
    groupDAO = new GroupDAO();
    membershipDAO = new MembershipDAO(userDAO, groupDAO);
    balanceDAO = new BalanceDAO();
    expenseDAO = new ExpenseDAO(groupDAO, membershipDAO);
    settlementDAO = new SettlementDAO(groupDAO, membershipDAO
    );
    cleanDatabase();
}

protected void cleanDatabase() throws SQLException {
    try (Statement stmt = connection.createStatement()) {
        stmt.execute("SET REFERENTIAL_INTEGRITY FALSE");
        stmt.executeUpdate("DELETE FROM settlements");
        stmt.executeUpdate("DELETE FROM balances");
        stmt.executeUpdate("DELETE FROM expense_participants"
    );
        stmt.executeUpdate("DELETE FROM expenses");
        stmt.executeUpdate("DELETE FROM memberships");
        stmt.executeUpdate("DELETE FROM groups");
        stmt.executeUpdate("DELETE FROM users");
        stmt.execute("SET REFERENTIAL_INTEGRITY TRUE");
    }
}
```

Listing 19: Setup e pulizia del database in BaseIntegrationTest

Questo approccio garantisce che ogni test parta da uno stato noto e controllato, eliminando il rischio di *flaky test* causati da dati residui di esecuzioni precedenti. La disabilitazione temporanea dei vincoli referenziali tramite `SETREFERENTIAL_INTEGRITYFALSE` consente la pulizia delle tabelle nell'ordine desiderato, garantendo l'**isolamento tra test** e prevenendo *flaky test* causati da dati residui.

5.2.2 UserServiceTest_UC1_UC2 (Autenticazione)

La classe collauda i flussi di registrazione e accesso al sistema. Il basic flow di UC1 è verificato dal test `UC1_signUp_withValidData_createsUser` , che controlla la corretta persistenza dell'utente. Gli alternative flow coprono: email duplicata, formato email non valido, login con email inesistente e login con password errata.

A livello di integrazione viene inoltre verificata la sicurezza della persistenza. Mentre la validazione dei dati avviene a livello di dominio, qui si

verifica che l'algoritmo di hashing venga effettivamente applicato prima del salvataggio nel database. Accedendo ai record bypassando deliberatamente la logica applicativa, si garantisce che le credenziali non siano mai archiviate in chiaro.

```
@Test
void UC1_signUp_passwordIsHashed() {
    String plainPassword = "mySecretPassword123";
    User user = userService.signUp("secure@test.com",
    plainPassword, "Secure User");

    // Accesso diretto al DB per verificare lo stato reale
    dei dati
    User fromDB = userDAO.findById(user.getUserId()).
    orElseThrow();

    assertNotEquals(plainPassword, fromDB.getPasswordHash());
    assertNotNull(fromDB.getPasswordHash());
    assertFalse(fromDB.getPasswordHash().isEmpty());
}
```

Listing 20: Verifica della sicurezza della persistenza bypassando il Service

5.2.3 GroupServiceTest (Configurazione e Gestione Membri)

I test di gruppo sono distribuiti in due classi distinte per area funzionale.

- `GroupServiceTest_UC12` collauda la configurazione strutturale e il rispetto dei vincoli di sicurezza relazionali. Vengono esercitate la creazione del gruppo (**UC3**), il join tramite codice (**UC4**) e l'aggiornamento atomico delle impostazioni (**UC12**). Il livello di integrazione assicura che un membro non-admin non possa forzare aggiornamenti sul database, verificando il corretto sollevamento della `UnauthorizedException`.

- `GroupServiceTest_UC10` si concentra sull'amministrazione dei membri. In questo contesto, le regole di business (già validate isolatamente nei test strutturali) vengono collaudate all'interno del flusso transazionale. Ad esempio, il vincolo che impedisce la rimozione di un membro con debiti pendenti viene qui esercitato per confermare che il `GroupService` interrompa l'operazione e prevenga modifiche indesiderate alle tabelle relazionali, applicando la regola indipendentemente dal ruolo del richiedente.

```
@Test
```

```
void UC10_removeMember_withDebt_shouldThrowBusinessException
    () {
     DomainException exception = assertThrows(DomainException.
    class, () -> {
         groupService.removeMember(groupId, debtorMembershipId
    , adminMembershipId);
     });

     // Si verifica che il Service abbia bloccato l'
    aggiornamento su DB
     assertTrue(isMemberActive(groupId, memberWithDebtId));
}
```

Listing 21: Verifica dell'integrità transazionale sulla rimozione membri

Un membro con credito può invece essere rimosso, garantendo che il credito rimanga storicizzato nel sistema.

5.2.4 ExpenseServiceTest_UC5_UC11 (Gestione Spese)

Questa suite valida la corretta propagazione degli eventi tramite il Pattern Observer all'interno del contesto transazionale. A differenza dei test unitari, qui si verifica la capacità del `ServiceLayer` di effettuare il *wiring dinamico*, ricollegando correttamente in memoria le entità lette dal database.

L'inserimento di una nuova spesa (**UC5**) deve innescare l'aggiornamento a cascata dei saldi, garantendo matematicamente a transazione conclusa il **vincolo di sistema chiuso**: la somma algebrica di tutti i balance estratti dal database deve essere pari a zero. Analogamente, per le **UC11**, si verifica che la **modifica dell'importo** inneschi un ricalcolo coerente e che la **cancellazione** (*soft delete*) istruisca gli Observer ad annullare i saldi riportandoli allo stato precedente.

```
@Test
void UC5_addExpense_updatesBalancesAutomatically() {
    expenseService.addExpense(group.getGroupId(),
        aliceMembership.getMembershipId(), new BigDecimal("
    100.00"),
        "Dinner", Category.FOOD,
        List.of(aliceMembership.getMembershipId(),
    bobMembership.getMembershipId()));

    Balance aliceBalance = balanceDAO.findByMembershipId(
    aliceMembership.getMembershipId()).orElseThrow();
    Balance bobBalance = balanceDAO.findByMembershipId(
    bobMembership.getMembershipId()).orElseThrow();

```

```
11      // Verifica dell'azione degli Observer post-persistenza
12      assertEquals(new BigDecimal("50.00"), aliceBalance.
        getAmount());
13      assertEquals(new BigDecimal("-50.00"), bobBalance.
        getAmount());
14
15      // Controllo del vincolo di sistema chiuso
16      BigDecimal total = aliceBalance.getAmount().add(
        bobBalance.getAmount());
17      assertEquals(BigDecimal.ZERO.setScale(2), total.setScale
        (2));
18  }
```

Listing 22: Collaudo del wiring dinamico e dell'aggiornamento saldi a cascata

5.2.5 BalanceServiceTest_UC6_UC8 (Saldi e Rimborsi)

In questa classe, oltre a validare l'output dell'algoritmo di minimizzazione dei debiti (`MinTransactionsStrategy`) su un set di dati persistito reale (**UC6**), l'attenzione è posta sulle transizioni di stato del database durante i rimborsi (**UC8**).

Viene esaminato in modo rigoroso il comportamento del sistema di fronte all'annullamento di un pagamento prima della sua conferma definitiva. Il test dimostra che, interrompendo il flusso, il database mantiene i `Balance` intatti e aggiorna unicamente lo stato della riga `Settlement` in `REJECTED`, confermando la corretta applicazione della regola architetturale della "doppia conferma".

```
1   @Test
2   void UC8_cancelSettlement_beforeConfirmation_shouldWork()
        throws Exception {
3       Settlement settlement = settlementService.
        createSettlement(
4           groupId, membershipId1, membershipId3, new BigDecimal
        ("20.00"));
5
6       settlementService.cancelSettlement(settlement.
        getSettlementId(), membershipId1);
7
8       // Si garantisce che non vi siano state scritture
        accidentali sui saldi
9       assertEquals(0, new BigDecimal("-50.00").compareTo(
        getBalance(membershipId1)));
10      assertEquals(0, new BigDecimal("80.00").compareTo(
        getBalance(membershipId3)));
11
```

```
12      Settlement fromDB = settlementDAO.findById(settlement.
        getSettlementId()).orElseThrow();
13      assertEquals(PaymentStatus.REJECTED, fromDB.getStatus());
14  }
```

Listing 23: Verifica della consistenza del database all'annullamento di un rimborso

Use Case	Flow	Test
UC1 Sign Up	Basic Flow	Registrazione con dati validi, hashing password
	Alternative 5a	Email duplicata
	Alternative 5b	Email/password non validi, nome vuoto
UC2 Login	Basic Flow	Login con credenziali valide
	Alternative 4a	Email inesistente, password errata
UC3 Create Group	Basic Flow	Creazione gruppo, assegnazione ruolo ADMIN
UC4 Join Group	Basic Flow	Join con codice invito, stato WAITING_ACCEPTANCE
	Alternative	Codice invito non valido
UC5 Add Expense	Basic Flow	Creazione spesa, aggiornamento saldi (Observer)
	Alternative 4a	Importo negativo/zero
	Alternative 5a	Descrizione vuota
UC6 View Balances	Basic Flow	Saldi corretti, MinTransactionsStrategy, isGroupSettled, getDebtors, getCreditors
UC8 Settle Debt	Basic Flow	Settlement completo e parziale
	Alternative 4a	Importo superiore al debito
	Alternative	Membro senza debiti, cancellazione settlement
UC9 Invite Member	Basic Flow	Generazione codice invito
	Alternative	Generazione da parte di non-admin
UC10 Manage Members	Basic Flow	Rimozione membro (saldo zero, in credito)
	Alternative	Rimozione membro con debiti, non-admin, admin con debiti, approvazione membership
UC11 Edit/Delete	Basic Flow	Edit/Delete + ricalcolo e annullamento saldi
	Alternative 2a/2b	Autorizzazioni (solo creator)
UC12 Configure Group	Basic Flow	Aggiornamento nome, descrizione, valuta
	Alternative	Modifica da parte di non-admin

Tabella 13: Copertura Use Case e Alternative Flows nei test di integrazione

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

Come anticipato durante l'analisi del Controller Layer, i test End-to-End (E2E) vengono eseguiti in modalità *headless*, ovvero senza coinvolgere direttamente la CLI interattiva.

Sfruttando il disaccoppiamento nativo dell'architettura, l'infrastruttura di test sostituisce il gestore reale dei menu con uno Stub dedicato, chiamato `StubNavigator`. Questo componente fittizio, iniettato nei Controller al

momento del setup, implementa l'interfaccia `Navigator` ma ne neutralizza il comportamento bloccante: invece di stampare a video e attendere input dall'utente, intercetta le chiamate di navigazione e registra semplicemente il proprio stato interno.

Questo approccio permette di esercitare e validare automaticamente l'intero flusso dei casi d'uso, mantenendo la logica applicativa isolata dal layer di input/output su console. Di seguito è riportata l'implementazione essenziale dello Stub utilizzato:

```
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
```

```
39      public void navigateToGroupDetails(Group group) {
40          this.currentPage = "GROUP_DETAILS";
41      }
42  }
```

Listing 24: Implementazione Stub per l'interfaccia Navigator