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 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.

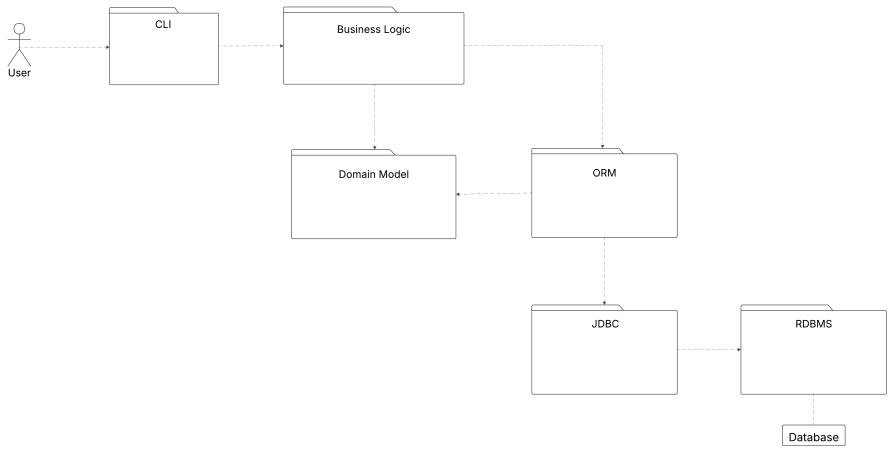

Figura 1: Panoramica dell'architettura logica a livelli del sistema.

2 Analisi dei Requisiti

2.1 Use Case Diagram

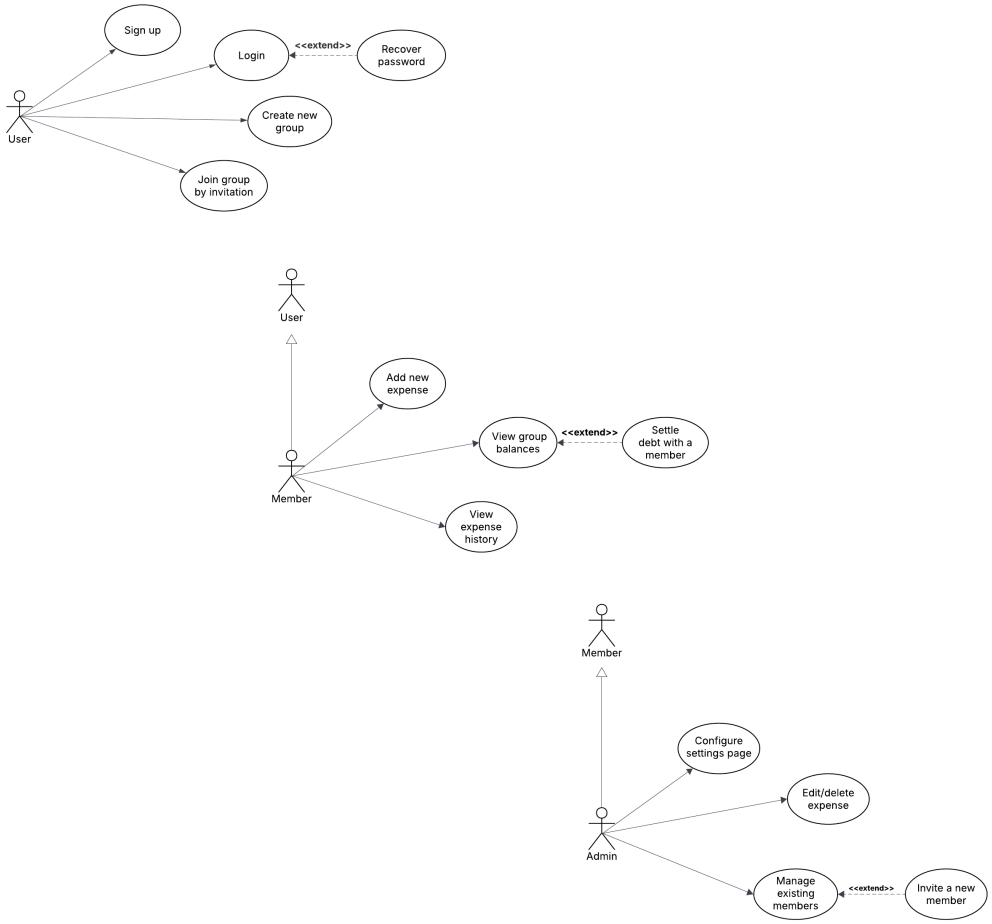

Figura 2: Use Case Diagram

2.2 Use Case Templates

UC1 – Sign Up	
Attore	User
Livello	Function
Pre-condizioni	L'utente non deve essere già registrato nel sistema.
Basic Course	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 #1).
Alternative Course	5a. Se l'email è già registrata, il sistema mostra un messaggio di errore "Account already exists". 5b. Se i campi obbligatori non sono compilati o le password non coincidono, il sistema richiede la correzione.

Tabella 1: UC1 – Sign Up

UC2 – Login	
Attore	User
Livello	Function
Pre-condizioni	L'utente deve essere già registrato nel sistema.
Basic Course	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 Course	4a. Se l'email o la password non sono corrette, il sistema mostra un messaggio "Invalid credentials" e permette un nuovo tentativo. 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 Course	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.
Alternative Course	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 Course	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. 5. Il sistema mostra un messaggio di conferma e reindirizza alla pagina del gruppo (**vedi Mockup #2**).
Alternative Course	3a. Se il codice è errato o scaduto, il sistema mostra "Invalid or expired invitation". 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 Course	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 e chiude il modale tornando alla pagina del gruppo.
Alternative Course	4a. Se l'importo non è valido o i campi obbligatori sono vuoti, il sistema mostra un messaggio di errore. 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 Course	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 – Lista di debiti/crediti reciproci ("chi deve a chi") 4. Il membro può visualizzare i dettagli delle transazioni passate.
Alternative Course	3a. Se non ci sono spese registrate, il sistema mostra "No expenses yet".

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 Course	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 Course	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 Course	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 e aggiorna immediatamente i saldi mostrati nella pagina.
Alternative Course	4a. Se il membro annulla l'operazione, il sistema chiude la richiesta senza registrare il pagamento.

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 Course	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. 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 Course	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 #). 4a. Se il link scade o viene revocato dall'Admin, il sistema mostra "The invitation has expired" (Test #).

Tabella 9: UC9 – Invite a New Member

UC10 – Manage Existing Members	
Attore	Admin
Livello	User Goal
Pre-condizioni	L'Admin è autenticato. Esiste almeno un gruppo attivo con uno o più membri.
Basic Course	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 Course	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" (Test #).

Tabella 10: UC10 – Manage Existing Members

UC11 – Edit/Delete Expense	
Attore	Admin
Livello	User Goal
Pre-condizioni	Esiste almeno una spesa registrata.
Basic Course	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 e la cronologia. 5. Il sistema notifica tutti i membri del gruppo.
Alternative Course	2a. Tentativo di modifica su spesa già saldata: il sistema mostra un messaggio di blocco. 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 Course	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 Course	3a. Errore di validazione (es. valuta non supportata). 3b. L'Admin annulla l'operazione: nessuna modifica viene applicata.

Tabella 12: UC12 – Configure Group Settings

2.3 Mockups

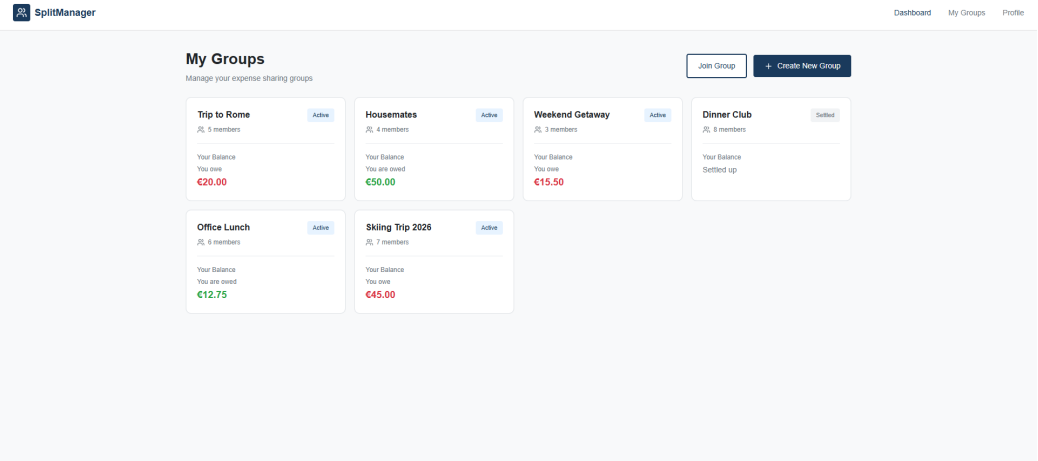

Figura 3: Mockup #1 – User Dashboard

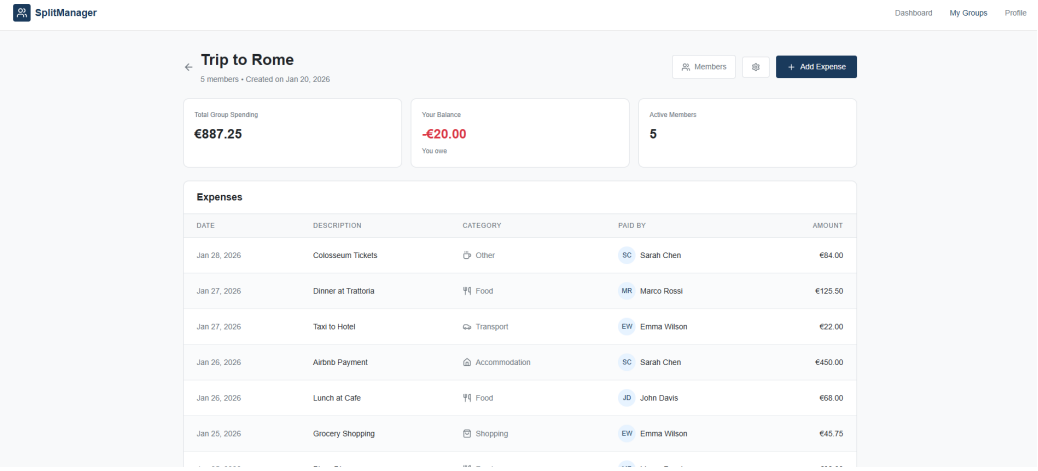

Figura 4: Mockup #2 – Group Details Page

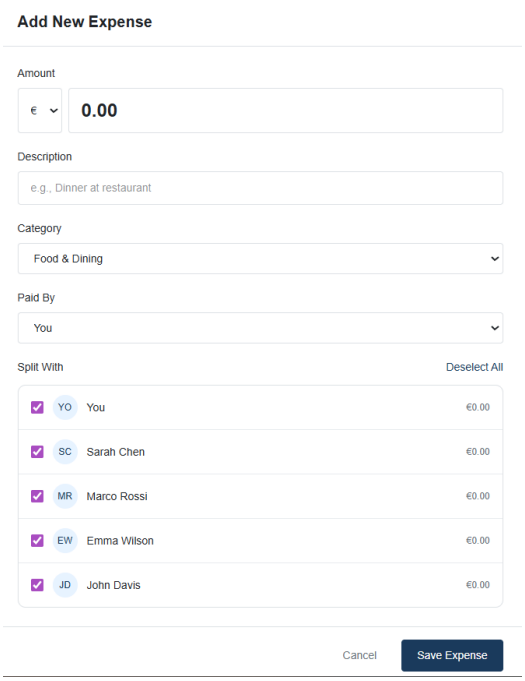

Figura 5: Mockup #3 – Add Expense Modal

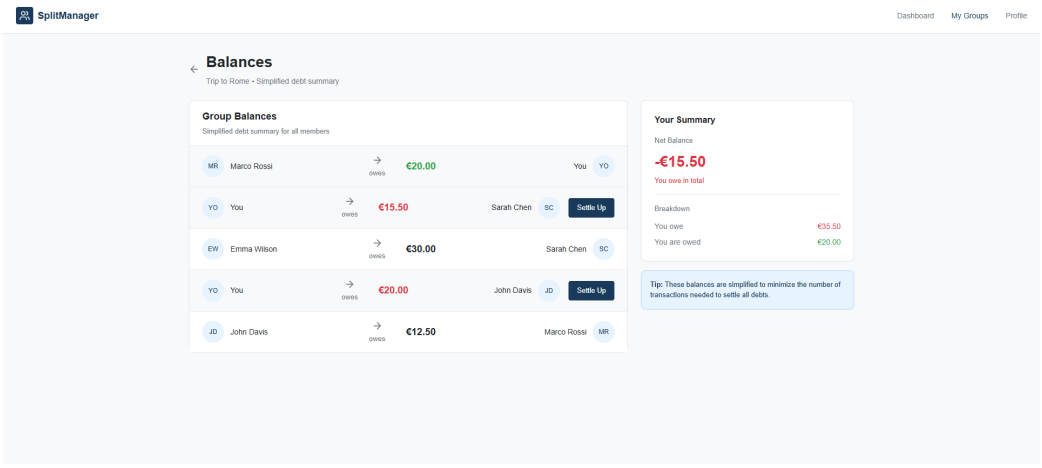

Figura 6: Mockup #4 – Balances & Settlement

2.4 Page Navigation Diagram

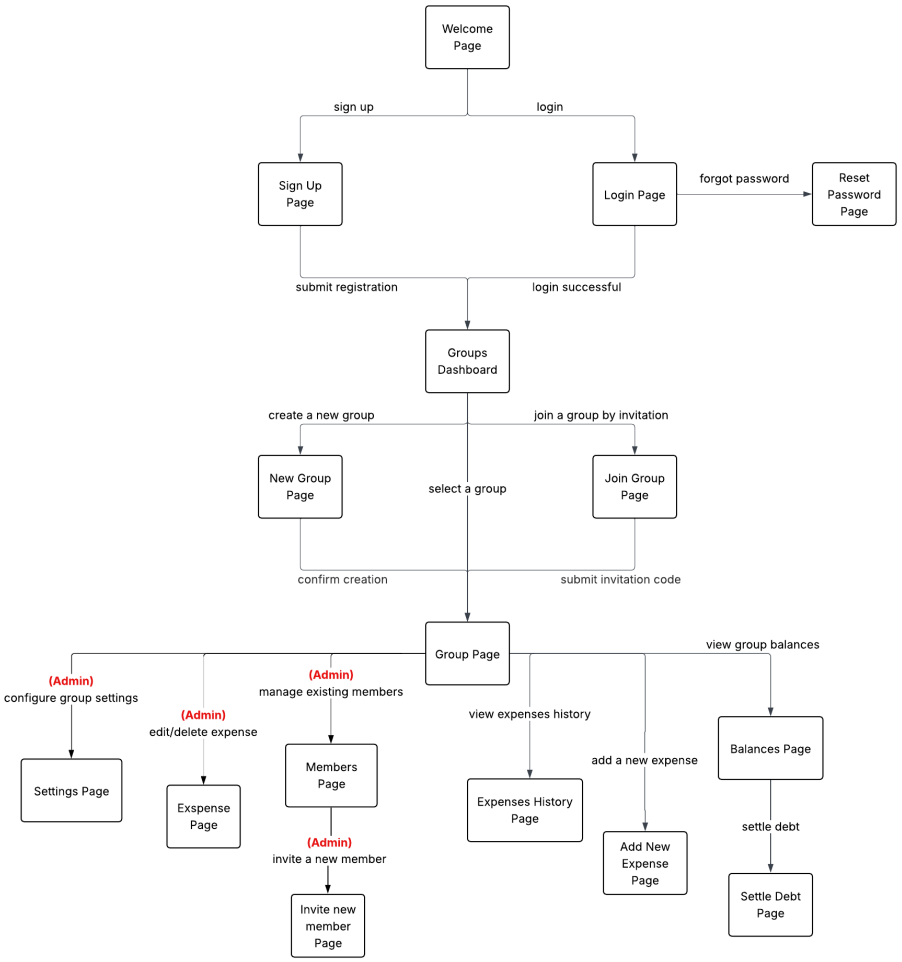

Figura 7: Page Navigation Diagram

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 layer incapsula la business logic ed è indipendente dagli altri livelli.

  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

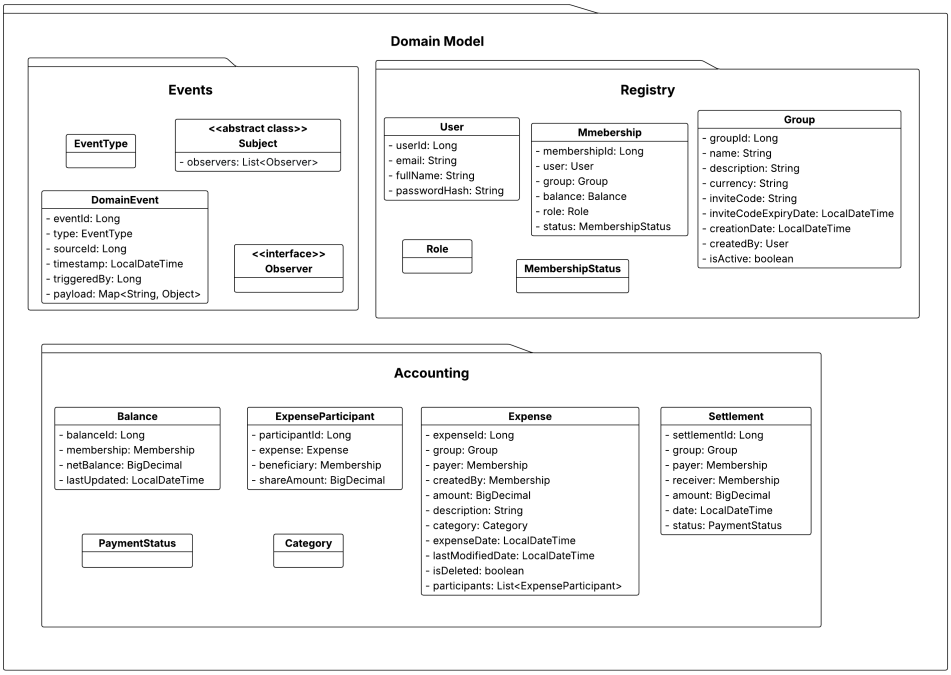

Figura 8: Suddivisione del Domain Model in sotto-package

- **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.

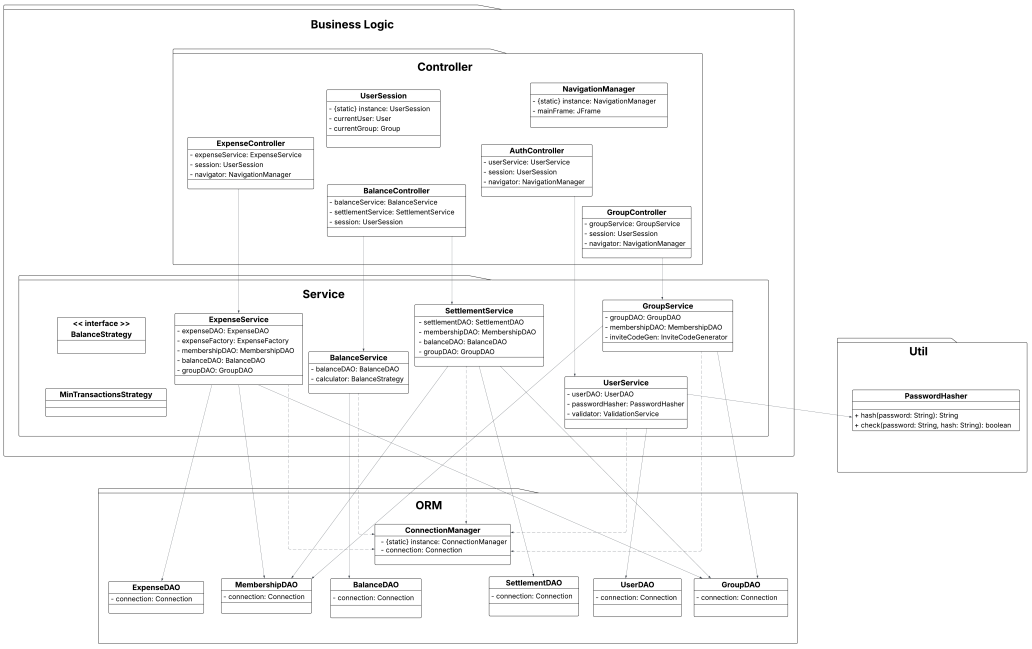

Figura 9: Dipendenze tra i package Controller, Service, ORM e Util

- **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.

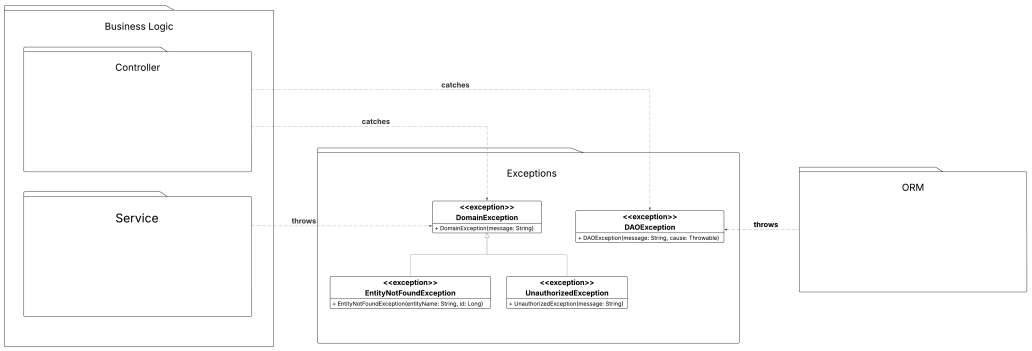

Figura 10: Exceptions Package

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:

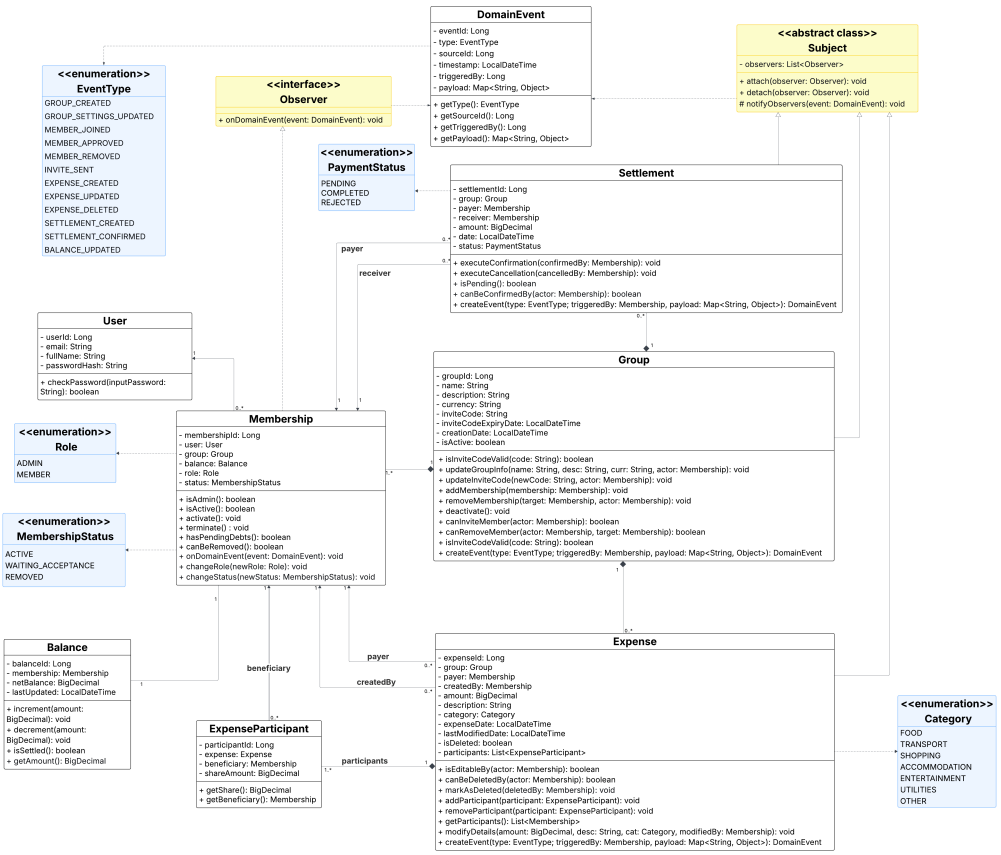

Figura 11: Domain Model Class Diagram

- **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

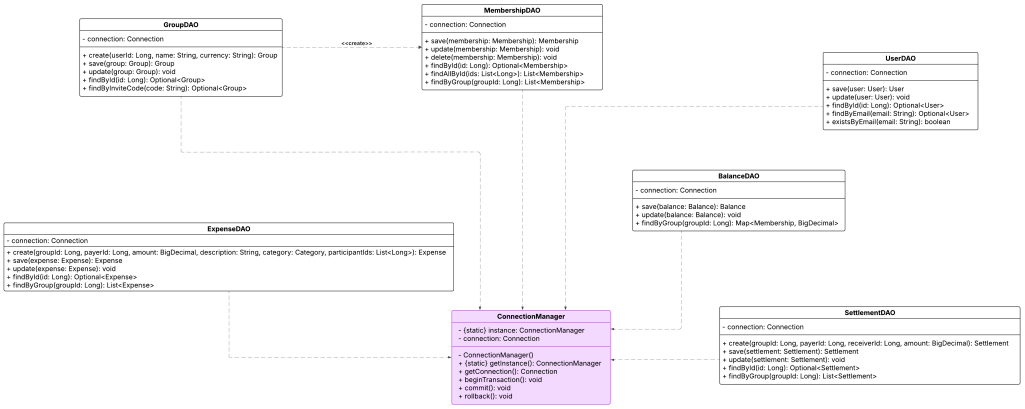

Figura 12: ORM Class Diagram

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.

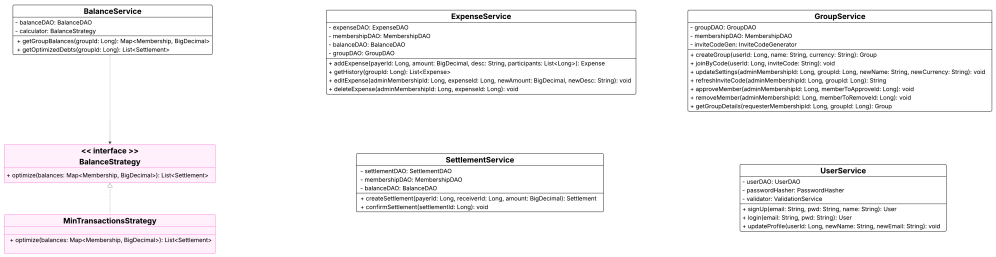

Figura 13: Service Layer Class Diagram

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

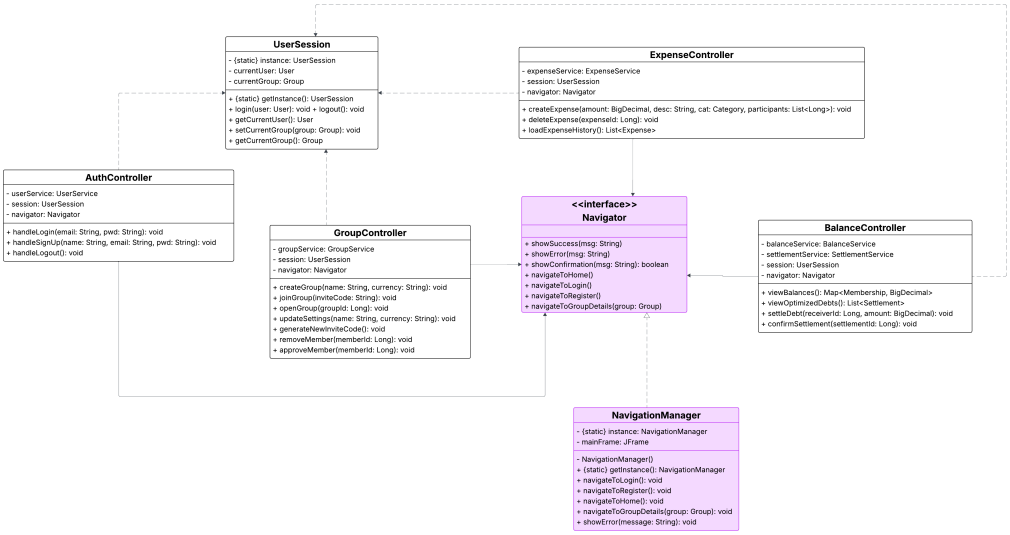

Figura 14: Controller Layer Class Diagram

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 14, 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

3.3.1 Modello ER

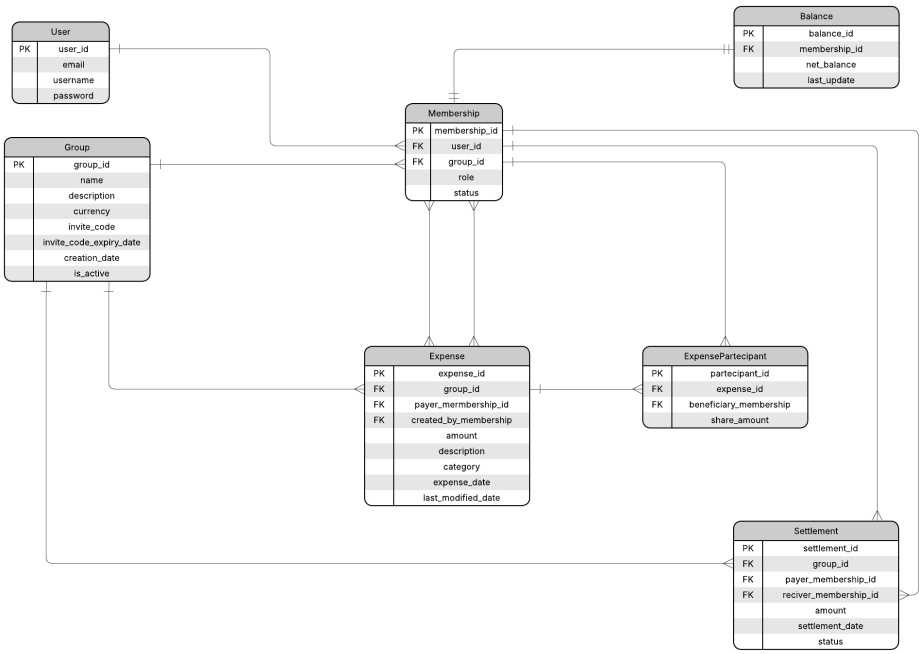

Figura 15: Modello ER

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.

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 16: 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.

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

Figura 17: Snippet 2 - ExpenseTest

```java
@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 18: 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.

```java
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 19: Snippet 4 - StubNavigator