Design Splitwise

Contents

System Design Steps

1. Overview of the System
2. Requirements Gathering
3. Clarify Requirements
4. Class Diagram
5. Schema Design
6. Code Implementation (Spring Boot)

1. Overview of the System

- Overview part you will have two cases
  a. You know the application.
    - If you know the application you will have to tell the basics that, this is what I understand from the application named Splitwise.
  b. You don't know the application.
    - If you don't know the application you will have to ask that, I don't know the application and I have not used Splitwise. Can you please elaborate in brief what do you mean by what kind of application we are building.

Assuming that we don't know Splitwise!

Interviewer will give you the problem statement.

Problem Statement: Splitwise System Design

- Splitwise is an **expense-sharing** and **tracking** application.
- Used when a **group of people** share expenses but pay at different times.

Example Scenario:

- Four friends (A, B, C, D) go to dinner and the total bill is ₹2000.
- Instead of everyone paying separately, **A pays the entire bill**.
- The fair share per person is **₹500** each.
- The app will track:

- **A has paid ₹2000** (₹1500 extra).
- **B**, **C**, and **D owe ₹500 each to A**.

- Over a long trip, multiple expenses occur, making manual tracking difficult.

Splitwise solves this problem by:

- Keeping track of expenses.
- Ensuring a minimal number of transactions for settlements.
- Providing a clear view of who owes whom.

2. Requirements Gathering & Clarify Requirements

1. User Management
   - Users must be able to register and create an account.
   - Users must be able to update their profile information.
   - Each user profile must contain at least:
     - Name
     - Phone Number
     - Password (for authentication)
2. Expense Management
   - Users can add an expense in one of the following ways:
     - Within a group (e.g., a trip or office team).
     - Among specific users (without a group).
   - Each expense must include:
     - Who paid what
     - Who owes what
     - Description of the expense
3. Group Management
   - Users can create groups for shared expenses.
   - Users can join and participate in groups.
   - Only the group creator has permission to:
     - Add or remove members.

- Users cannot query groups they are not part of (RBAC - Role-Based Access Control).

4. Expense Tracking
   - A user can see their total owed amount (how much they owe or are owed).
   - A user can see a history of all their expenses.
   - A user can see a history of group expenses for any group they are part of.
5. Settle-Up (Debt Simplification)
   - Individual Settle-Up:
     - A user can request a settle-up to clear their debts.
     - The system will suggest a list of transactions that, when executed, will ensure the user no longer owes or is owed money.
   - Group Settle-Up:
     - A user can request a settle-up for a group they belong to.
     - The system will suggest optimized transactions to settle all debts within that group.
     - Note: Expenses outside the group are not included in group settle-up.
6. Optimized Debt Settling
   - The application should minimize the number of transactions required to settle up within a group.

3. Identify Classes

- The two strategies for creating a class diagram are:
  1. **Identifying Nouns** – Extracting relevant entities (nouns) from requirements or problem statements to determine the potential classes.
     - Check if we need to store the data for this noun in the table or not.
  2. **Visualization** – This includes understanding the system flow, identifying key interactions, and mapping user journeys.

Identified Core Classes

- User
  - Attributes: `userId`, `name`, `phoneNumber`, `encryptedPassword`
- Expense
  - Attributes: `expenseId`, `description`, `totalAmount`, `createdBy`, (map)`paidBy`, (map) `owedBy`
- Group
  - Attributes: `groupId`, `groupName`, `adminId`, `members`, `expenses`
- ExpenseType
  - `REAL`, `DUMMY`

Reverse Expense

Understanding the Concept of Reverse Expense

- A **Reverse Expense** is an approach where transactions between users are not stored directly in the database. Instead, the system dynamically calculates and updates the balances of users.
- This eliminates the need to explicitly store transactions, reducing database storage and improving efficiency.

Why Not Store Transactions Directly?

- Normally, when a transaction occurs (e.g., one person paying another), you might think of storing it in a "Transactions" table. However, this might not always be necessary. The reasoning is:
  - Transactions are **not permanent** records until the payment actually happens.
  - Users only need to **see** how much they owe or are owed.
  - The system can dynamically **compute balances** without persisting each transaction.
  - If a payment is made, the balances can be **adjusted using a reverse expense**, rather than storing the transaction itself.

Example Scenario: Group Dinner Expense

- Consider four friends—**A, B, C, and D**—going for dinner, where **A pays the full amount of ₹2000**. The cost is split equally, meaning each person should pay **₹500**.

User	Paid Amount	Share	Extra Paid (Paid - Share)
A	₹2000	₹500	+₹1500
B	₹0	₹500	-₹500
C	₹0	₹500	-₹500
D	₹0	₹500	-₹500

- Now, **C wants to settle up** by paying **₹500 to A**.
- Instead of storing a "**C paid A ₹500**" transaction, we update the balances:
  - **C had -₹500 (owed ₹500), so after paying, C's balance becomes ₹0**.
  - **A had +₹1500 (paid extra ₹1500), after receiving ₹500, A's balance reduces to ₹1000**.
- Updated table after C pays A:

User	Paid Amount	Share	Extra Paid (Updated)
A	₹2000	₹500	+₹1000
B	₹0	₹500	-₹500
C	₹500	₹500	₹0
D	₹0	₹500	-₹500

How Reverse Expense Works

- Instead of explicitly storing the transaction, we **add an opposite entry** (a reverse expense) in the balances:
  - **C pays ₹500 → Reduce C's negative balance by ₹500** (add +500 to C).
  - **A receives ₹500 → Reduce A's extra paid balance by ₹500** (subtract 500 from A).

C++ Classes

1. The `User` class

```
namespace Splitwise::Model {
   class User {
      int userId;
      string name;
      string phoneNumber;
      string encryptedPassword;
   public:
      User(int id, string name, string phone, string password);
   };
}
```

2. The `Expense` class

```
namespace Splitwise::Model {
   using namespace std;
   using namespace std::chrono;
   class Expense
   {
      int expenseId;
      string description;
      double totalAmount;
      time_point<system_clock> createdAt;
      shared_ptr<User> createdBy;
      ExpenseType expenseType;
      vector<shared_ptr<UserExpense>> userExpenses;
   public:
      Expense(int id, string desc, double amount,
         shared_ptr<User> creator, ExpenseType type);
   };
}
```

3. The `Group` class

```
namespace Splitwise::Model {
   using namespace std;
   class Group {
      int groupId;
      string groupName;
      shared_ptr<User> admin;
      vector<shared_ptr<User>> members;
      vector<shared_ptr<Expense>> expenses;
   public:
      Group(int id, string name, shared_ptr<User> adminUser);
      void addMember(shared_ptr<User> user);
      void addExpense(shared_ptr<Expense> expense);
   };
}
```

4. The `UserExpense` `class`

- Why is the UserExpense class needed?
    - **Mapping Relationships in a Database**
        - `User` and `Expense` are separate entities.
        - A user may **pay for an expense** or **owe a share of an expense**.
        - The relationship between them includes attributes like **amount paid, amount owed, and currency**.
    - **Avoiding Direct Storage of Maps in Relational Databases**
        - SQL databases do **not support direct storage** of complex data structures like maps (`std::map` or `std::unordered_map`).
        - Instead of storing a `map<User, Amount>`, a separate **mapping table** is created to handle this relationship.
    - **Scalability and Extensibility**
        - If tomorrow we need to support multiple currencies, timestamps, or additional attributes, adding them to a separate table (`UserExpense`) is **easier and cleaner**.
        - Directly modifying the `Expense` table for each new feature would lead to frequent schema changes, affecting existing data.
    - **Normalization (Avoiding Unnecessary Columns in Expense Table)**
        - If we stored `amount_paid` and `amount_owed` as columns in `Expense`, handling multiple users for one expense would become difficult.
        - A **separate mapping table (`UserExpense`)** allows multiple users to be associated with a single expense.

```
namespace Splitwise::Model {
   class UserExpense {
      int userId;
      int expenseId;
      double amount;
      UserExpenseType userExpenseType;
   public:
      UserExpense(int userId, int expenseId,
         double amount, UserExpenseType type);
   };
}
```

4. Schema Design

- Schema design follows a structured process after creating the **Class Diagram**. The steps involved are:

1. Create Tables for Each Class
    - Every class in the **Class Diagram** is converted into a table in the database.
2. Define Columns for Each Table
    - **Primitive attributes** (e.g., integers, strings) are directly added as columns in their respective tables.
    - **Non-primitive attributes** (e.g., objects, collections) require Cardinality analysis.
3. Determine Cardinality and Apply Cardinality Rules
    - **Cardinality** defines how many instances of one entity relate to another entity.
    - Cardinality rules dictate how foreign keys and relationships are structured in the database.
    - **One-to-One:** ID of one side is stored as a foreign key on the other side.
    - **One-to-Many or Many-to-One:** ID of the 'one' side is stored as a foreign key on the 'many' side.
    - **Many-to-Many:** A **mapping table** is created to store relationships between entities.

Tables with primitive attributes

- The `Users` table
    - The Users table has only primitive attributes.

userId	name	phoneNumber	encryptedPassword

```
CREATE TABLE USER (
    USERID INT PRIMARY KEY,
    NAME VARCHAR(255) NOT NULL,
    PHONENUMBER VARCHAR(20) UNIQUE NOT NULL,
    ENCRYPTEDPASSWORD VARCHAR(255) NOT NULL
);
```

- The `Expenses` table
    - The `Expenses` table with only primitive attributes…

expenseId	description	totalAmount	createdAt		

- The `UserExpenses` table
    - The `UserExpenses` table with only primitive attributes…

id	amount		

- The `UserExpenseType` table
    - The corresponding class is an enum and hence it will have only id and value.

typeId	typeName

```
CREATE TABLE USEREXPENSETYPE (
    TYPEID INT PRIMARY KEY,
    TYPENAME VARCHAR(50) UNIQUE NOT NULL
)
```

- The `Groups` table
    - The `Groups` table with only primitive attributes…

groupId	groupName		

- The `ExpenseType` table
    - The corresponding class is an enum and hence it will have only id and value.

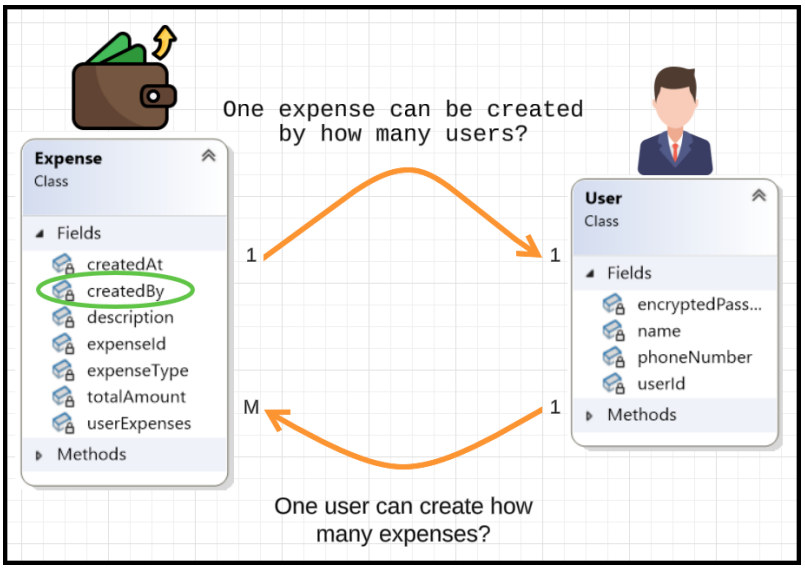
typeId	typeName

```
CREATE TABLE EXPENSETYPE (
    TYPEID INT PRIMARY KEY,
    TYPENAME VARCHAR(50) UNIQUE NOT NULL
)
```

Finding the Cardinality

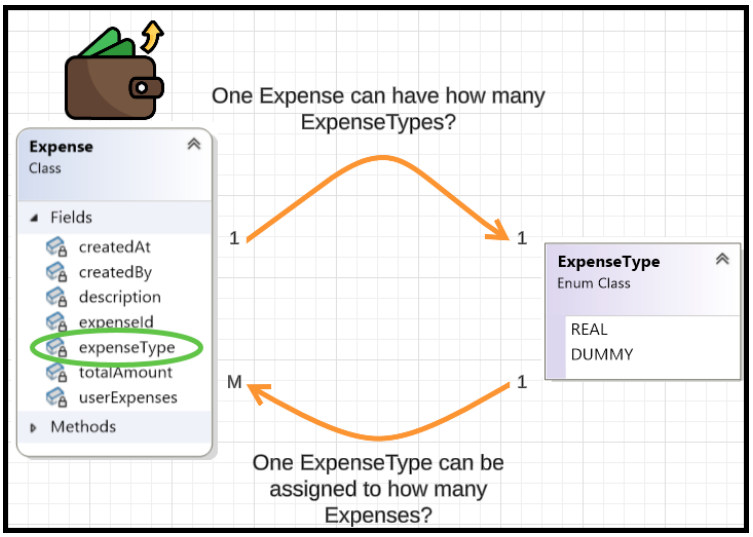
- Let's consider all the classes again and find the Cardinality between these classes.

1. `Expense` and `User`
   - The `Expense` **`class`** has a non-primitive attribute **`createdBy`** of type `User`.



   - ID of '1' side on 'M' side.
   - So, '`Expenses`' table should have `Users` table `ID` column. Let's add it to '`Expenses`' table.

2. `Expense` and `ExpenseType`
   - The `Expense` **`class`** has a non-primitive attribute **`expenseType`** of type `ExpenseType`.

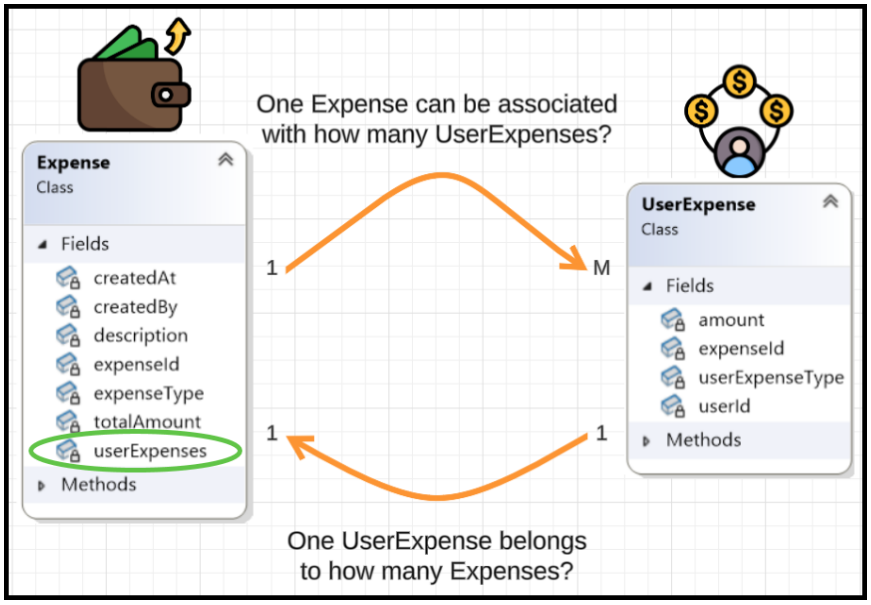


- ID of '1' side on 'M' side.
- So, '`Expenses`' table should have **`ExpenseType`** table `ID` column. Let's add it to '`Expenses`' table.
- The `Expenses` Table

expenseId	description	totalAmount	createdAt	createdBy	expenseTypeId

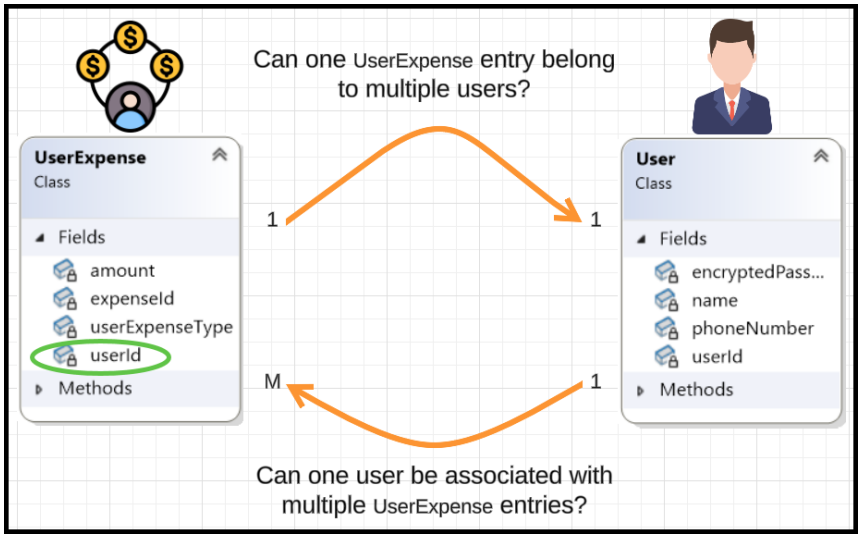
```
CREATE TABLE EXPENSE (
    EXPENSEID INT PRIMARY KEY,
    DESCRIPTION VARCHAR(255),
    TOTALAMOUNT DOUBLE,
    CREATEDAT TIMESTAMP,
    CREATEDBY INT NOT NULL,
    EXPENSETYPEID INT NOT NULL,
    FOREIGN KEY (CREATEDBY) REFERENCES USER(USERID),
    FOREIGN KEY (EXPENSETYPEID) REFERENCES EXPENSETYPE(TYPEID)
);
```

3. `Expense` and `UserExpense`
   - The `Expense` **`class`** has a non-primitive attribute **`userExpenses`** of type `vector<shared_ptr<UserExpense>>`.
   - The relationship between `Expense` and `UserExpense` is **one-to-many (1:M)** because:
     - **One** `Expense` **can have multiple** `UserExpense` **records** (since multiple users may be involved in the same expense).
     - **Each** `UserExpense` **entry belongs to exactly one** `Expense` (it records how much a specific user owes or has paid for that particular expense).



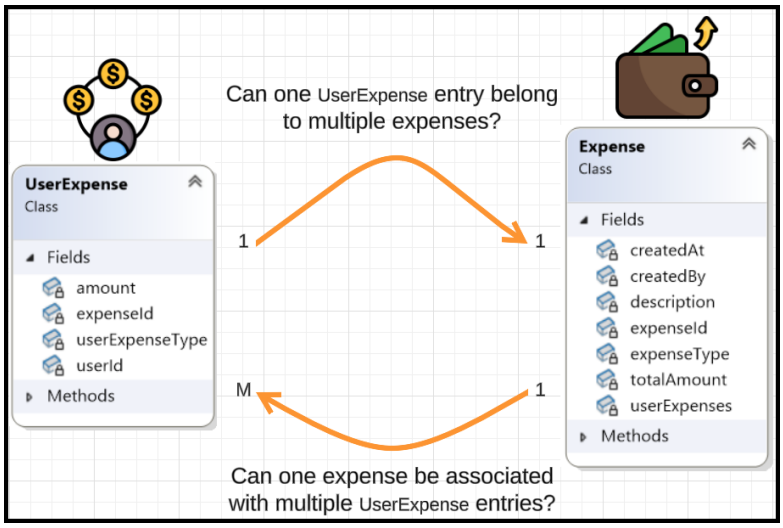
- ID of '1' side on 'M' side.
- So, 'UserExpenses' table should have Expenses table ID column. Let's add it to 'UserExpenses' table.

4. UserExpense and User
   - The UserExpense class has a primitive attribute **userId** of type int which is the ID of User.
   - **Even though userId is an integer**, we still need to determine the cardinality because:
     - **Foreign Key Relationships:** The **userId** in UserExpense refers to a User, so we need to understand the logical association.
     - **Database Design:** Knowing the cardinality ensures we apply the correct **foreign key constraints** (e.g., FOREIGN KEY (userId) REFERENCES User(userId)).
     - **Query Optimization:** Cardinality helps in designing efficient **indexes** and **queries**.
   - **One** UserExpense **entry belongs to exactly one** User (since each expense share is associated with a specific user).
   - **One** User **can be part of multiple** UserExpenses (since a user can be involved in multiple expenses).



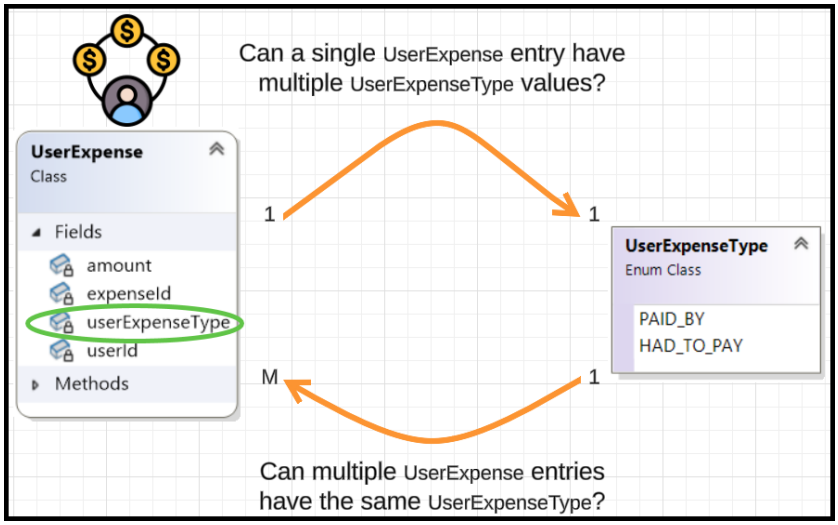
- ID of '1' side on 'M' side.
- So, `userId` of `Users` table in the '`UserExpenses`' table. Let's add it to '`UserExpenses`' table.

5. `UserExpense` and `Expense`
   - The `UserExpense` `class` has a primitive attribute **`expenseId`** of type `int` which is the ID of `Expense`.
   - **Even though `expenseId` is an integer**, we still need to determine the cardinality.
   - **One `Expense` can have multiple `UserExpense` entries** (since an expense is split among multiple users).
   - **One `UserExpense` entry belongs to exactly one `Expense`** (each `UserExpense` entry represents a user's share in a particular expense).



- ID of '1' side on 'M' side.
- So, `expenseId` of `Expenses` table in the '`UserExpenses`' table. Let's add it to '`UserExpenses`' table.

6. `UserExpense` and `ExpenseType`
   - The `UserExpense` `class` has a non-primitive attribute **`userExpenseType`** of type `UserExpenseType`.
   - Understanding the Relationship
     - `UserExpense` represents how an expense is split among users.
     - Each `UserExpense` has a `UserExpenseType`, which can be either:
       - `PAID_BY`: The user paid for the expense.
       - `HAD_TO_PAY`: The user owes money for the expense.
   - Cardinality Analysis
     - One `UserExpense` has exactly one `UserExpenseType` (1:1).
     - One `UserExpenseType` (`PAID_BY` or `HAD_TO_PAY`) can be used by multiple `UserExpense` entries (M:1).



   - ID of '1' side on 'M' side.
   - So, **`userExpenseTypeID`** of **`UserExpenseType`** table in the '`UserExpenses`' table. Let's add it to '`UserExpenses`' table.

- The UserExpenses table

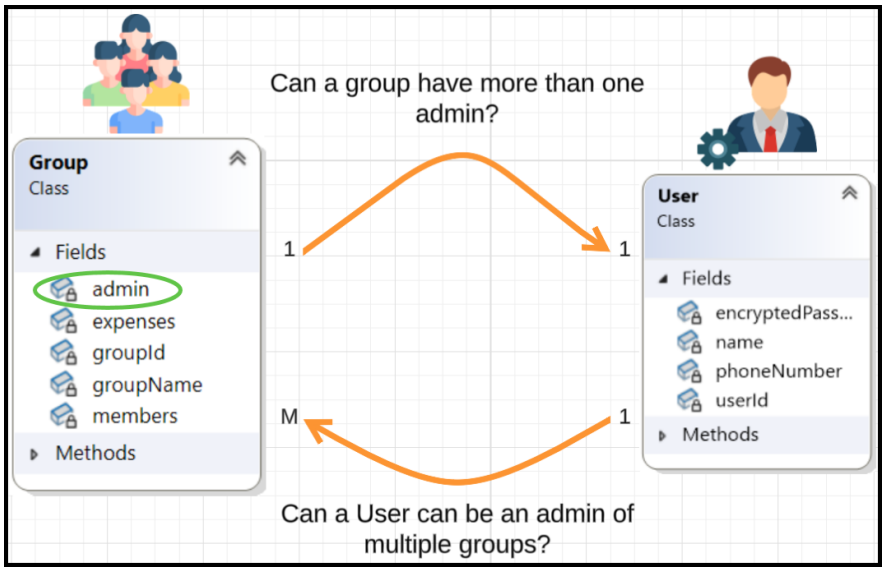
id	amount	userId	expenseId	userExpenseType

```
CREATE TABLE USEREXPENSES (
    ID INT NOT NULL,
    AMOUNT DOUBLE NOT NULL,
    USERID INT NOT NULL,
    EXPENSEID INT NOT NULL,
    USEREXPENSETYPE INT NOT NULL,

    PRIMARY KEY (ID),

    FOREIGN KEY (USERID) REFERENCES USERS(USERID) ON DELETE CASCADE,
    FOREIGN KEY (EXPENSEID) REFERENCES EXPENSES(EXPENSEID) ON DELETE CASCADE,
    FOREIGN KEY (USEREXPENSETYPE) REFERENCES USEREXPENSETYPE(TYPEID) ON DELETE RESTRICT
);
```

7. The `Group` **`class`** and `User` **`class`**
   - The `Group` **`class`** has a non-primitive attribute **`admin`** of type `User`.



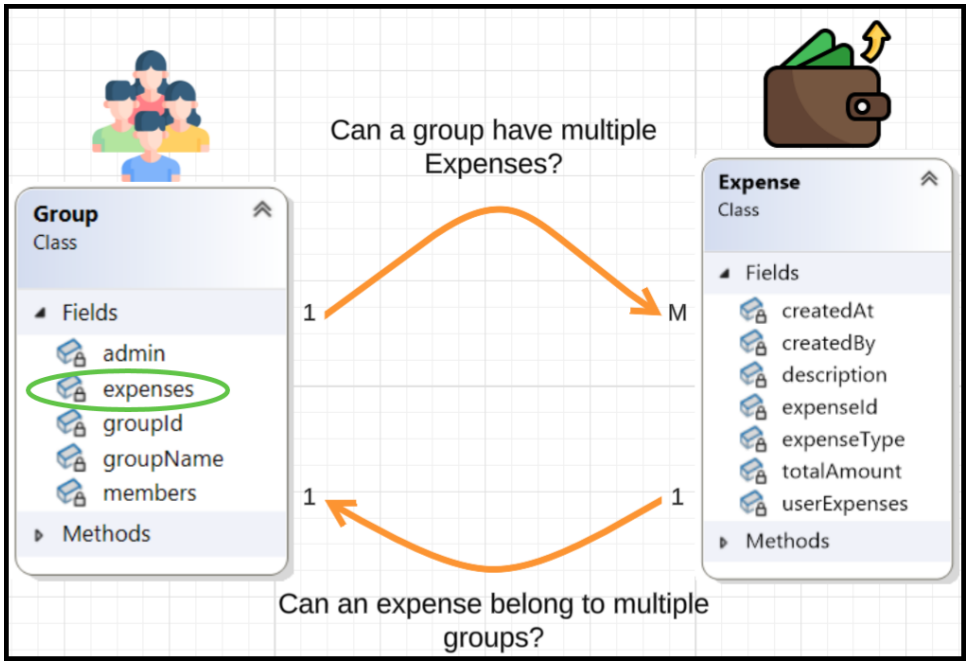
   - ID of '1' side on 'M' side.
   - So, `userId` of `Users` table in the '`Groups`' table as `adminId`. Let's add it to '`Groups`' table.

- First let's create 'Groups' table.

groupId	groupName	adminId

```
CREATE TABLE GROUPS (
    GROUPID INT PRIMARY KEY,
    GROUPNAME VARCHAR(255) NOT NULL,
    ADMINID INT NOT NULL,  -- ADMIN IS A USER
    FOREIGN KEY (ADMINID) REFERENCES USERS(USERID)
);
```

- The `Group class` has a non-primitive attribute **`expenses`** of type `vector<shared_ptr<Expense>>`.



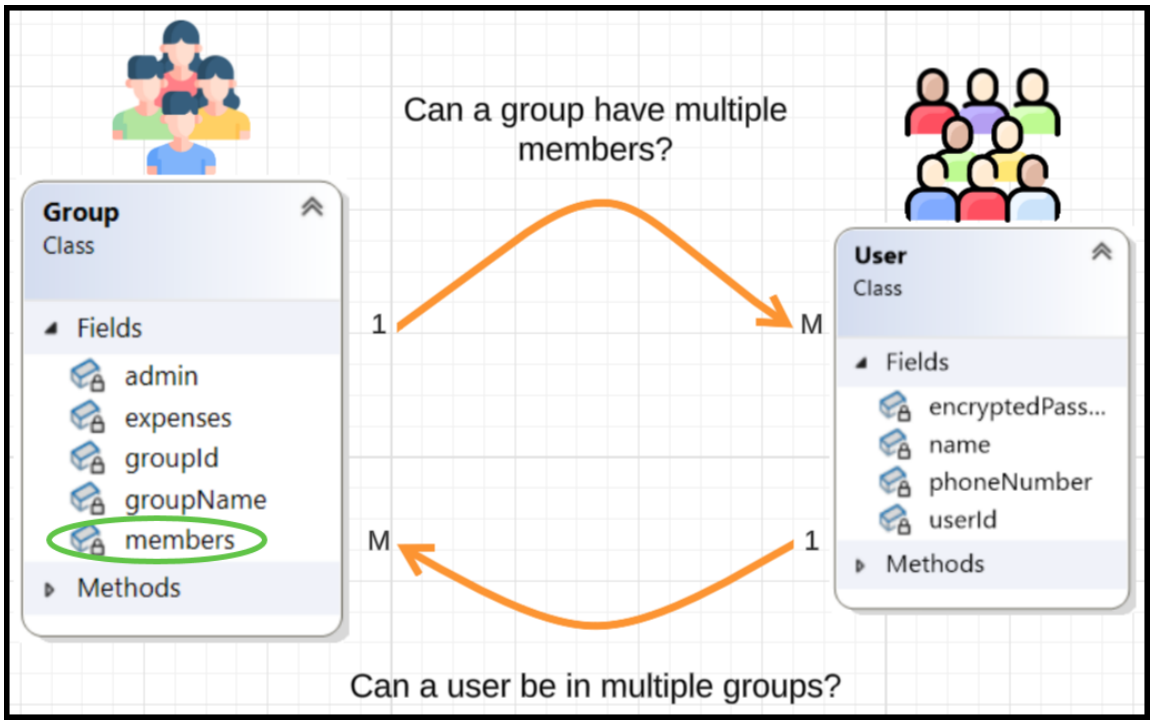
- ID of '1' side on 'M' side.
- So, `groupId` of `Groups` table in the '`Expenses`' table. Let's add it to '`Expenses`' table. So, let's update the '`Expenses`' table.

expenseId	description	totalAmount	createdAt	createdBy	expenseType	groupId

```sql
CREATE TABLE EXPENSES (
    EXPENSEID INT PRIMARY KEY,
    DESCRIPTION VARCHAR(255) NOT NULL,
    TOTALAMOUNT DOUBLE NOT NULL,
    CREATEDAT TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    CREATEDBY INT NOT NULL,
    EXPENSETYPE INT NOT NULL,
    GROUPID INT NOT NULL,

    FOREIGN KEY (CREATEDBY) REFERENCES USERS(USERID) ON DELETE CASCADE,
    FOREIGN KEY (GROUPID) REFERENCES GROUPS(GROUPID) ON DELETE CASCADE
);
```

- The `Group` `class` has a non-primitive attribute **`members`** of type `vector<shared_ptr<User>>`.



- Confirms **M:M** between `Group` and `User`.
- Hence, we need to create a mapping table.
- The `USER_GROUP` mapping table

userId	groupId

```sql
CREATE TABLE USER_GROUP (
    USERID INT NOT NULL,
    GROUPID INT NOT NULL,
    PRIMARY KEY (USERID, GROUPID),
    FOREIGN KEY (USERID) REFERENCES USERS(USERID) ON DELETE CASCADE,
    FOREIGN KEY (GROUPID) REFERENCES GROUPS(GROUPID) ON DELETE CASCADE
);
```

Problem with the EXPENSES table

- We initially designed the `EXPENSES` table to include a `groupId` column, assuming every expense belongs to a `GROUP`.

expenseId	description	totalAmount	createdAt	createdBy	expenseType	groupId

- However, after analysing the data:
    - **1 million expenses exist in the system.**
    - **Only 1 lakh expenses (10%)** are associated with a group.
    - **9 lakh expenses (90%)** are individual expenses and do not belong to any group.
- This means **90% of** `groupId` **values would be NULL**, making the table a **sparse table** (a table with many NULL values).

Solution: Use a Mapping Table

- Instead of keeping `groupId` in the `EXPENSES` table, we should:
    - **Remove** `groupId` **from** `EXPENSES`.

```sql
CREATE TABLE EXPENSES (
    EXPENSEID INT PRIMARY KEY,
    DESCRIPTION VARCHAR(255) NOT NULL,
    TOTALAMOUNT DOUBLE NOT NULL,
    CREATEDAT TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    CREATEDBY INT NOT NULL,
    EXPENSETYPE INT NOT NULL,

    FOREIGN KEY (CREATEDBY) REFERENCES USERS(USERID) ON DELETE CASCADE
);
```

expenseId	description	totalAmount	createdAt	createdBy	expenseType

- **Create a separate** `EXPENSE_GROUP` **table** to map expenses to groups.
- The EXPENSE_GROUP mapping table.

expenseId	groupId

```
CREATE TABLE EXPENSE_GROUP (
    EXPENSEID INT NOT NULL,
    GROUPID INT NOT NULL,

    PRIMARY KEY (EXPENSEID, GROUPID),
    FOREIGN KEY (EXPENSEID) REFERENCES EXPENSES(EXPENSEID) ON DELETE CASCADE,
    FOREIGN KEY (GROUPID) REFERENCES GROUPS(GROUPID) ON DELETE CASCADE
);
```