

Splitting Bills, Simplified: Developing a Splitwise-like App with Java and SOLID Principles



Choudharynishantplawat · Follow

6 min read · 3 days ago





Introduction

In today's world of shared expenses — whether it's roommates splitting rent, friends sharing a meal, or colleagues handling a joint project — **expense-sharing apps** have become indispensable. **Splitwise** is one of the most popular apps for tracking shared expenses, but how would you build such a system from scratch?

This article walks you through creating a **Splitwise-like expense-sharing app** in Java, ensuring **scalability**, **maintainability**, and **efficiency** using the **SOLID principles** and **design patterns**. We'll cover the end-to-end process, including gathering requirements, designing the system, implementing key components, and how it all fits into a clean architecture.

Understanding the Requirements

To design a Splitwise-like app, we need to focus on the following requirements:

Core Features:

1. **User Management:** Users can add, track, and settle expenses.
2. **Expense Tracking:** Users can log expenses with other users or in groups.

3. **Expense Splitting:** Expenses can be split equally, by percentage, or based on custom amounts.
4. **Balance Calculation:** The system tracks who owes whom and how much.
5. **Debt Settlement:** Users can settle debts between each other.

Non-Functional Requirements:

- **Scalability:** The app should handle a large number of users and groups.
- **Concurrency:** Multiple users may update expenses at the same time.
- **Security:** Sensitive data (e.g., user balances) must be secure.
- **Maintainability:** The system should be easy to extend with new features.

Approach: SOLID Principles & Design Patterns

To create a robust system, we'll adhere to the **SOLID principles**:

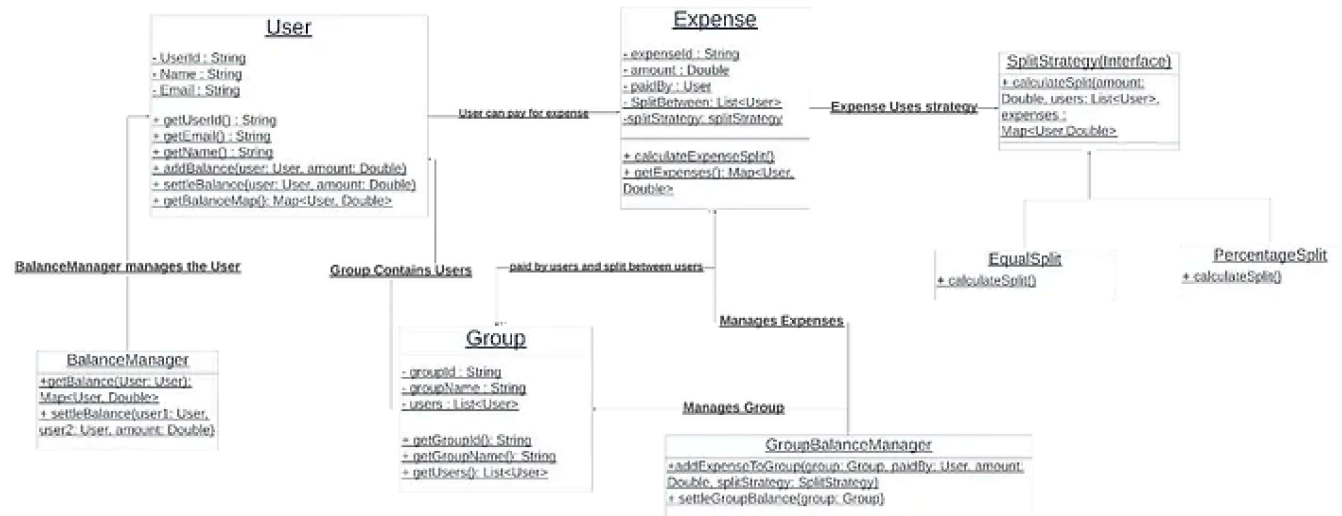
1. **Single Responsibility Principle (SRP):** Each class should have one responsibility. For example, the `User` class manages user data, and the `Expense` class manages the expense logic.
2. **Open/Closed Principle (OCP):** Classes should be open for extension but closed for modification. We achieve this by using the **Strategy Pattern** to allow different splitting methods (equal, percentage, etc.).

3. **Liskov Substitution Principle (LSP):** Subtypes must be substitutable for their base types. We ensure that all split strategies can be used interchangeably.
4. **Interface Segregation Principle (ISP):** No client should be forced to implement interfaces they don't use. The splitting strategies implement a simple interface specific to their behavior.
5. **Dependency Inversion Principle (DIP):** High-level modules should not depend on low-level modules. Both should depend on abstractions. We'll use **interfaces** and **factories** to decouple high-level logic from specific implementations.

In addition, we'll leverage design patterns like the **Strategy Pattern** (for splitting expenses), **Factory Pattern** (for creating expense objects), and the **Singleton Pattern** (for managing services like `BalanceManager`).

System Design: UML Diagram

Here's how the components relate to each other in the system:



Code Implementation

Let's walk through some key parts of the code, applying the SOLID principles and design patterns discussed earlier.

1. User Class (SRP)

The `User` class manages user-specific details and balances between users:

```

package main.java.model;
import java.util.*;
import java.math.*;
public class User {
    private String userId ;
    private String name ;
    private String email ;
    private Map<User, Double> balanceMap ;
  
```

```
public User(String userId, String name, String email){
    this.userId = userId ;
    this.name = name ;
    this.email = email;
    this.balanceMap = new HashMap<>();
}

public String getUserId(){
    return userId ;
}

public String getName(){
    return name ;
}

public String getEmail(){
    return email ;
}

public void addBalance(User user, double amount){
    balanceMap.put(user, balanceMap.getOrDefault(user, 0.0)+amount);
}

public Map<User, Double> getBalancemap(){
    return balanceMap;
}

public void settleBalance(User user, double amount){
    if(balanceMap.containsKey(user)){
        double currentBalance = Math.abs(balanceMap.get(user));
        double newBalance = currentBalance-amount;
        if(newBalance == 0.0){
            balanceMap.remove(user);
        }else{
            balanceMap.put(user, newBalance);
        }
    }
}
}
```

2. Expense and Split Strategy (OCP)

The `Expense` class uses the **Strategy Pattern** to allow flexible splitting strategies (e.g., equally or by percentage):

```
package main.java.model;
import java.util.*;

public class Expense {
    private String expenseId ;
    private double amount ;
    private User paidBy ;
    private List<User> splitBetween ;
    private SplitStrategy splitStrategy ;
    private Map<User, Double> expenses;

    public Expense(String expenseId, double amount, User paidBy, List<User> splitBetween, SplitStrategy splitStrategy) {
        this.expenseId = expenseId;
        this.amount = amount ;
        this.paidBy = paidBy;
        this.splitBetween = splitBetween;
        this.splitStrategy = splitStrategy;
        this.expenses = new HashMap<>();
    }

    public void calculateExpenseSplit(){
        splitStrategy.calculateSplit(amount, splitBetween, expenses);
    }

    public Map<User, Double> getExpenses(){
        return expenses;
    }
}
```



```
}
```

SplitStrategy Interface

```
public interface SplitStrategy {  
    void calculateSplit(double amount, List<User> users, Map<User, Double> expenses)  
}
```

Equal Split Strategy

```
public class EqualSplit implements SplitStrategy {  
    public void calculateSplit(double amount, List<User> users, Map<User, Double> expenses)  
    {  
        double splitAmount = amount / users.size();  
        for (User user : users) {  
            expenses.put(user, splitAmount);  
        }  
    }  
}
```

Percentage Split Strategy

```
package main.java.model;

import java.util.List;
import java.util.Map;

public class PercentageSplit implements SplitStrategy {
    private Map<User, Double> percentageMap;

    public PercentageSplit(Map<User, Double> percentageMap){
        this.percentageMap = percentageMap;
    }

    @Override
    public void calculateSplit(double amount, List<User> users, Map<User, Double> expenses) {
        for (User user : users){
            double percentage = percentageMap.get(user);
            double userAmount = amount*(percentage/100);
            expenses.put(user,userAmount);
        }
        System.out.println("Expenses splitted by Percentage");
    }
}
```

3. Group Class

The `Group` class encapsulates a group of users who share expenses. This class holds the group's details and the list of users in the group.

```
package main.java.model;
import java.util.*;
public class Group {
    private String groupId ;
    private String groupName ;
    private List<User> users;

    public Group(String groupId, String groupName, List<User> users){
        this.groupId = groupId ;
        this.groupName = groupName;
        this.users = users;
    }

    public String getGroupId() {
        return groupId;
    }

    public String getGroupName() {
        return groupName;
    }

    public List<User> getUsers() {
        return users;
    }
}
```

4. BalanceManager (ISP, DIP)

The `BalanceManager` handles retrieving and settling balances between users, ensuring high-level business logic is separated from the underlying implementation:



```
public class BalanceManager {
```

```
    public void settleBalance(User user1, User user2, double amount) {  
        user1.addBalance(user2, -amount);  
        user2.addBalance(user1, amount);  
    }  
}
```

5. GroupBalanceManager

The **GroupBalanceManager** handles the business logic for adding an expense to the group and updating the balances of all users in the group accordingly.

```
package main.java.service;  
import main.java.model.*;  
import java.util.*;  
  
public class GroupBalanceManager {  
    public void addExpenseToGroup(Group group, User paidBy, double amount, Split  
Expense ex = new Expense("E003", amount, paidBy, group.getUsers(), split  
ex.calculateExpenseSplit();  
    Map<User, Double> splitExpenses = ex.getExpenses();  
    //update the balance with all users  
    for(User user : group.getUsers()){  
        if(user != paidBy){  
            System.out.println("User : "+user.getName()+" paidBy : "+paidBy.
```

```
        paidBy.addBalance(user, splitExpenses.get(user));
        user.addBalance(paidBy, -splitExpenses.get(paidBy));
    }
}
}
```

6. Main Class: Bringing it All Together

The `Main.java` file demonstrates how the application works in practice. We'll create users, a group, and expenses to see how balances are updated.

Main.java: Running the Expense Sharing Application

```
package main.java;
import java.util.Arrays;
import java.util.Map;

import main.java.model.*;
import main.java.service.BalanceManager;
import main.java.service.GroupBalanceManager;

public class Main {
    public static void main(String args[]){
        //create users
        User user1 = new User("001", "Nishant", "nishant@gmail.com");
        User user2 = new User("002", "Amit", "amit@gmail.com");
        User user3 = new User("003", "rohit", "rohit@gmail.com");
        //create Group
```

```

Group group = new Group("gr001", "Expense Management", Arrays.asList(user1, user2, user3));
// create equal split expense
SplitStrategy equalSplit = new EqualSplit();
GroupBalanceManager groupBalanceManager = new GroupBalanceManager();
groupBalanceManager.addExpenseToGroup(group, user1, 100, equalSplit);

//check balance for all the users
BalanceManager balanceManager = new BalanceManager() ;
Map<User, Double> balanceUser2Map = balanceManager.getBalance(user2);
System.out.println("checking if : "+balanceUser2Map.size());
for(User user : balanceUser2Map.keySet()){
    System.out.println("User : "+user.getName()+" amount : "+balanceUser2Map.get(user));
}
System.out.println("User1 Balance: " + balanceManager.getBalance(user1));
System.out.println("User2 Balance: " + balanceManager.getBalance(user2));
System.out.println("User3 Balance: " + balanceManager.getBalance(user3));

//settle balance between user1 and user2
user2.settleBalance(user1, 33.33);
System.out.println("User2 Balance after settlement: " + balanceManager.getBalance(user2));
}
}

```

Conclusion

In this article, we've successfully built a **Splitwise-like expense-sharing system** using Java, focusing on the **SOLID principles** and design patterns like **Strategy** and **Singleton**. We also discussed how the **GroupBalanceManager** handles group-level expenses and ensures balances are correctly tracked.

This approach ensures the system is:

- **Scalable:** We can easily add new features (like different splitting strategies).
- **Maintainable:** Code is separated into well-defined components (e.g., `User`, `Expense`, `GroupBalanceManager`).
- **Extensible:** We can introduce more advanced features like concurrency handling, database integration, or even mobile/web interfaces.

About the Author

Nishant Choudhary is a Software Developer with around 2 years of experience in Software Development where he worked in multiple organizations and contributed to solving different kinds of complex problems with his creativity and enthusiasm for learning and different kinds of skills such as problem-solving, web designing and development, analysis of architecture and development of Progressive Web Applications.

[Java Development](#)[Design Patterns](#)[Solid Principles](#)[System Design Interview](#)[Software Engineering](#)



Written by Choudharynishantplawat


Follow

2 Followers

Software Engineer sharing insights and tips on coding, technology trends, and best practices in software development.

More from Choudharynishantplawat




 Choudharynishantplawat

Java + Popcorn = Build Your Own BookMyShow in a Day! 🎬🍿

No Hollywood CGI here, just plain old Java (and probably a lot of caffeine).


Oct 8 🖱️ 52

 Choudharynishantplawat

From Arthur to Joker: Taming Chaos with SAGA and CQRS...

Exploring how the Joker's chaotic transformation mirrors the power of SAGA...


Oct 10

 Choudharynishantplawat

Fastify vs. Express: The Ultimate Battle for Web Framework...

Fastify is like the superhero of web frameworks, swooping in to save the day wit...

Jul 22 🖱️ 51

 Choudharynishantplawat in Stackademic

Why Jest is So Last Year: Embrace Node.js's Native Test Runner...

Remember when testing in Node.js was like trying to catch a greased pig? 🐷👉 Well,...

Jul 31 🖱️ 50 💬 1



See all from Choudharynishantplawat

Recommended from Medium



 Nidhi Jain  in Code Like A Girl

7 Productivity Hacks I Stole From a Principal Software Engineer

Golden tips and tricks that can make you unstoppable

★ Oct 15 🖱️ 2.4K 💬 48 📌⁺



 DesignNerds in Level Up Coding

Low level design: URL Shortener System

Scalable and Enterprise-Ready Low-Level Design of a URL Shortener Service:...

★ Oct 12 🖱️ 6 📌⁺

Lists



General Coding Knowledge

20 stories · 1672 saves



Stories to Help You Grow as a Software Developer

19 stories · 1439 saves



Leadership

61 stories · 468 saves



Good Product Thinking

12 stories · 731 saves

Flipkart



 Sahil Kumar

Backend Engineer —at Flipkart Interview Experience

I recently went through the Flipkart Backend Engineer interview, and the experience was...

★ Sep 19 🖱 150 💬 1 📌



 Brian Jenney

3 Lessons from the Smartest Developers I've Worked With

I have a confession.

Oct 11 🖱 3.5K 💬 54 📌



 The Java Trail

Mastering Apache Kafka for Financial Transactions: Ensuring...



 Sylvain Tiset

Demystifying Garbage Collection Algorithms

Apache Kafka has emerged as a go-to solution for handling large-scale, high-...

★ 4d ago 🖱 8



A lot of programming languages are using a Garbage Collector (GC) to manage memory....

Sep 4 🖱 102



See more recommendations

[Help](#) [Status](#) [About](#) [Careers](#) [Press](#) [Blog](#) [Privacy](#) [Terms](#) [Text to speech](#) [Teams](#)