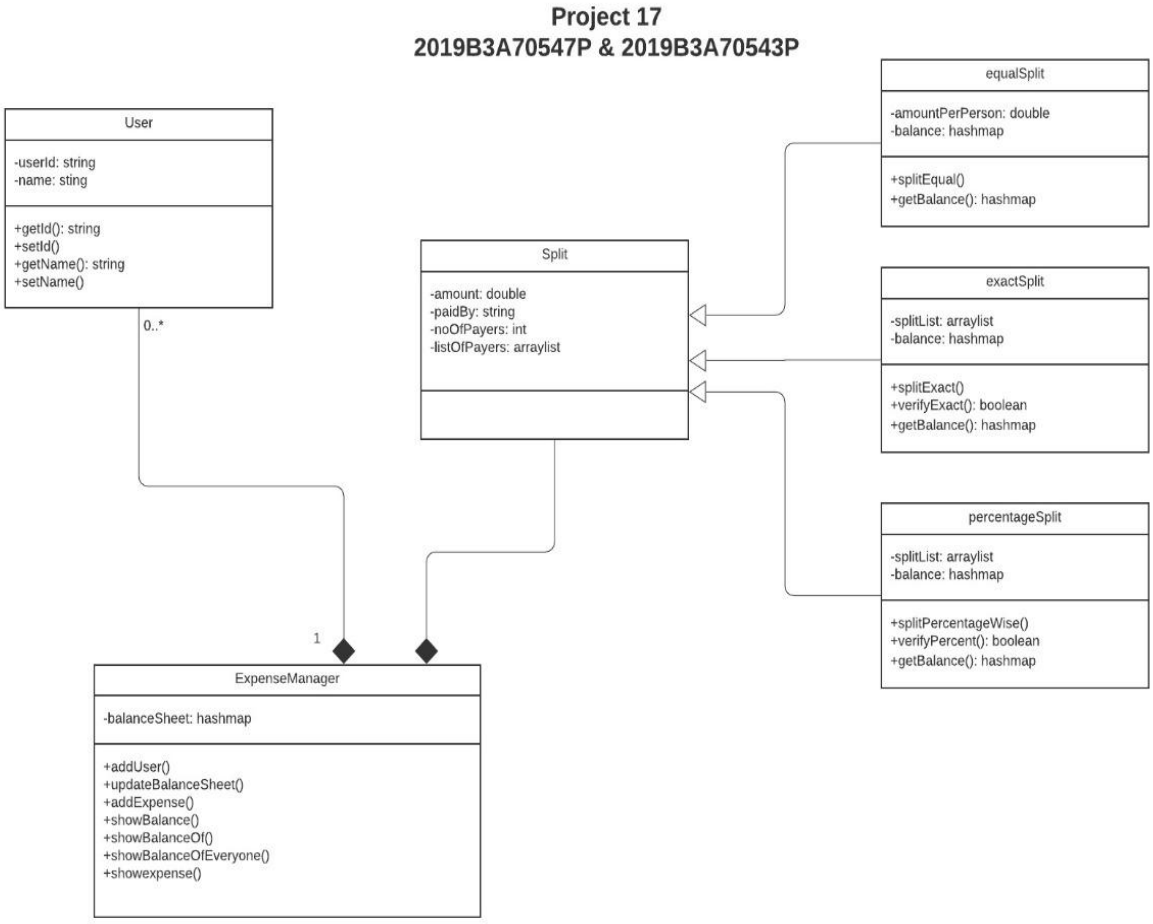
UML class diagram



DESIGN PATTERNS AND OOPS PRINCIPALS :

1. Observer Pattern :

In our program we are adding expense using expensemangener class by calling “addexpense” methods whenever we are adding expense we are also calling update method which is updating balance of all users and also updating balancesheet .**We are coding to concrete implementations, not interfaces** and we have to change our balancesheet whenever a new user is entered in our program and **We haven’t encapsulated what changes** . so we can use observer pattern to solve this problem .

This pattern could be used in a way like making “ExpenseManager" as observable and “user” as an observer interface where we can register and deregister multiple users (user 1, user2 ,.....) . We can add expenses in the system through the expensemanager and it should notify observers and update their balance , which could be displayed whenever the user wanted to print them .

ExpenseManager as subject interface with adduser , removeuser and updatebalance method could be implemented here and user can be implemented as observer interface and we can display his record from here.

2. **Strategy Pattern :**

   We had not exactly implement this pattern but we have use **inheritance to implement split behaviour** , we had created a class split and further extended 3 class from split class.

   If we had studied this pattern earlier we could have implemented our code in way like creating a expense class with with data members like who paid , no of users , list of users amount to be paid , and type of expense type (in current case equal,exact, percent) etc .and then we would have been created 3 inherited expense classes then we could have separate thing which are similar from those which are different like split behaviour in every expense class should have been implemented differently . so **we could implement a interface splitbehaviour . 3 concrete strategy for split behaviou**r 1. Equal split 2. Exact split 3.percent split

   In this way we could have implement our code using strategy pattern and follow some of principal of oops like **abstraction of interface** , **encapsulated split behaviours.**

3. Decorator Pattern :

Not recommended this kind of pattern for given program .

4. Factory Method :

In our program inside splitwise class in order to split the expense between users we need to create different types of split classes like EQUAL , EXACT , PERCENT and to implement the same we have used various if-else loops inside our main splitwise class . In order to avoid this we could have use Factory design pattern and should have created a single creator class ("splitcreate" class) and a single product class split() which return a split class based on user input , i.e if users want to split expense equally then object of equalsplit must be returned and if user want to split expense Percent wise than a object of percentsplit must be split. But now new implementation will result in violation on one of the oops principal '**Open for extension closed for modification' .**

So we can use the factory method in our program for creating instances of different split classes to make our code more efficient .

5. Singleton Pattern :

This pattern could be used for implementation of our program . We could use this pattern to implement the expensemanger class as we want only one instance of expensemanger and global point of access and we do not want users to create multiple objects/instances of class "expensemanager" because if multiple objects of expensemanage are created that would result in creation of multiple balance sheets , which will create confusion and can result in addition of incorrect records in our program . Applying this pattern could be very useful in implementing the GUI as it is very easy to maintain all records if only a single balance sheet(expensemanfer object) is being created .