Ding YuChen- Project Portfolio

1. Overview

This portfolio documents my contributions to PalPay, a software engineering project under the module, CS2103T Software Engineering. PalPay is an expenditure tracking application. The user interacts with it using a CLI, and it has a GUI created with JavaFX. It is written in Java, and has about 10 kLoC

For the project, my main role was to implement the Split features, which allow the user to record when bills are split between friends and when they settle up.

2. Summary of contributions

- **Code contributed**: RepoSense
- **Major enhancement**: Implemented the `Ledger` tab and functions associated with it.
  - What it does:
    - Allow user to split a bill easily between people. This can be by shares, or evenly, with or without including the user.
    - Allow individuals to settle up.
  - Justification: In our target demographic, many express that keep track of transactions between friends. Our application aims to solve this pain point by providing a simple way to keep track of debts.
  - Highlights: The `Ledger` tab has a split screen user interface, for the user to keep track of unpaid dollars and each transaction recorded between friends. The `Ledger` will only display people with outstanding balances on the **left** side of the screen, while transactions are listed on the **right**. (See below)
- **Minor enhancement**: added checks and improved error messages for handling `Amount`.
- **Other contributions**:
  - Project management:
    - Authored and assigned multiple issues. (Issues: #226, #199, #197, #136)
    - Reviewed and merged pull requests. (Pull requests: #70, #133)
    - Fixed numerous bugs. (Examples: #186, #185)
  - Enhancements to existing features:
    - Refactor `Transactions` into *Operations*, such that the `Model` can be overloaded to achieve polymorphism. (Pull requests: #95)
    - Refactor `VersionedBankAccount` into `VersionedUserState` that allows for the integration of the `Ledger` class, without associating the balances of `BankAccount` and `Ledger`. (Pull request: #133)

        - Wrote comprehensive unit test cases for the `Split` and `Receive` commmands and parsers. (Pull requests: #70, #114, #202, #205)
    - Documentation:
        - Reformatted the User Guide for a more sequential flow.
        - Wrote the details for the following commands in the User Guide.
            - Split
            - Receive
        - Created UML diagrams to help in the explanation of `split` and `receive` commands in the Developer Guide.
    - Community:
        - Reported bugs and suggestions for other teams in the module. (Examples: T14-2 #194, T14-2 #195, T12-4 #208, T12-4 #207, T12-4 #202)

3. Contributions to the User Guide

Given below are sections I contributed to the User Guide. They showcase my ability to write documentation targeting end-users.

3.1. Splitting a Bill with Friends : `split`

Split a bill with your friends
Format: `split $/AMOUNT n/NAME1 a/DESCRIPTION [d/DATE] [n/NAME2]… [s/SHARE]…`

- `DESCRIPTION` is prefixed with `a/`, unlike for other commands
- `[SHARE]` defines portion of bill to be paid by each person
    - if no shares are given, `AMOUNT` will be split evenly across all people, including user
    - you are included in the bill if number of shares is **1** more than number of people
        - your share of the bill will be the first listed share
    - each person's share is assigned in order
        - i.e. last person's share is the last share listed

CAUTION	Shares can be 0 but result is not guaranteed to be meaningful

Ledger GUI

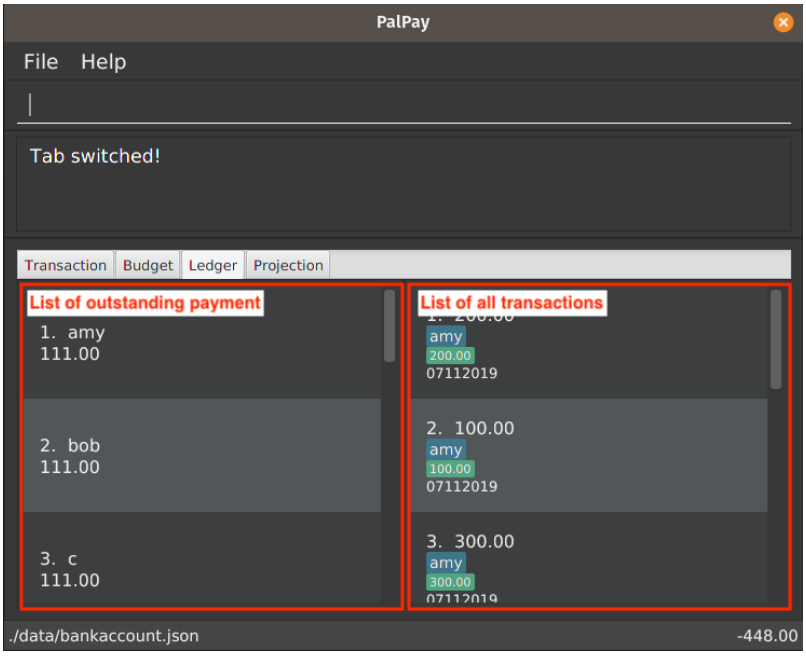

Figure 1. Sample Ledger Graphical User Interface

This is how the *Ledger* looks when you switch to the splits tab.
The left shows the people who has unresolved balances with you, while the right lists all transactions that have to do with the *Ledger*.
Ledger's balance is separate from PalPay's balance. It is displayed in the same position, at the bottom right corner.

:INFORMATION_SOURCE:	`split` **does not** include how much you spent into the *Ledger* balance.

3.1.2. Example Usage

- `split $/1000 n/Amy n/Betty n/Catherine n/Dan a/haidilao`

```
$1000 is split equally between Amy, Betty, Catherine, Dan and the user.
```

1. Enter appropriate command into the command line.

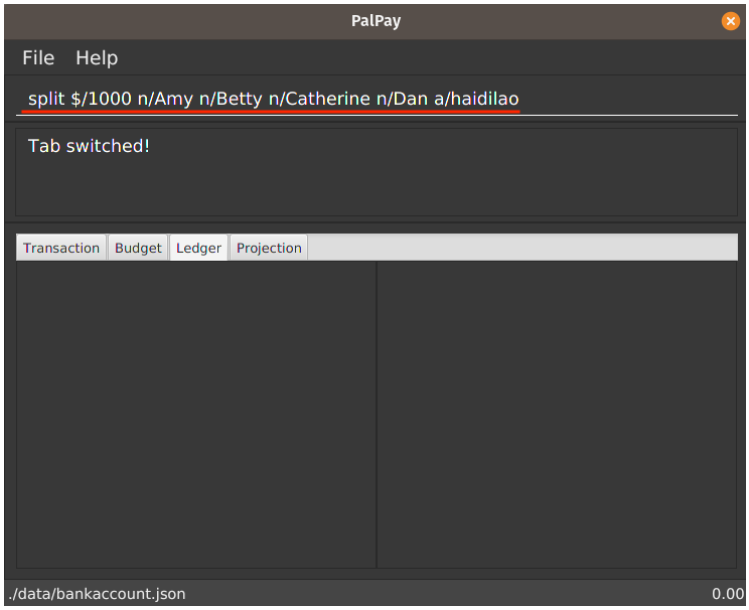

Figure 2. Input for Splitting Evenly

2. Result is displayed accordingly

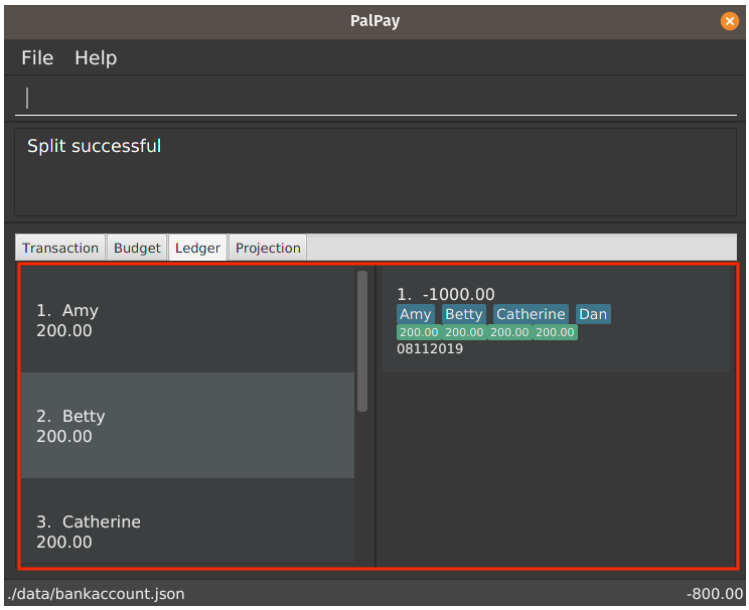

Figure 3. Amy, Betty, Catherine and Dan owes $200 each

For an even split of $1000, each person pays $200. Therefore *Ledger* shows $200 on the tab of each person. *Ledger* balance **does not** include how much you spend. In this bill, one is owed $800 in total from the rest of his friends. Therefore *Ledger* balance is -$800, as shown in the bottom right.

- `split $/100 n/Albert n/Bernard n/Clement s/2 s/1 s/7 a/kbbq dinner`

```
$100 is split with Albert owing $20, Bernard owing $10 and Clement owing $70.
```

1. Enter appropriate command into the command line.

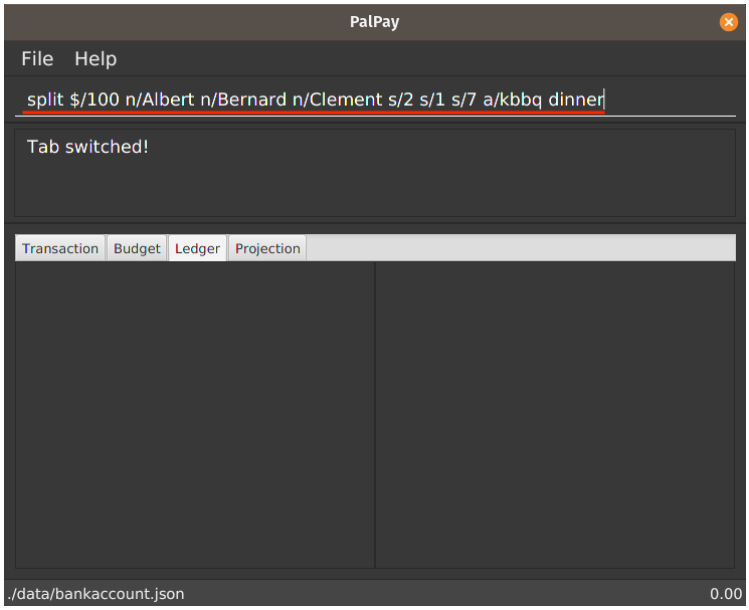

Figure 4. Input for Splitting Unevenly

2. Result is displayed accordingly

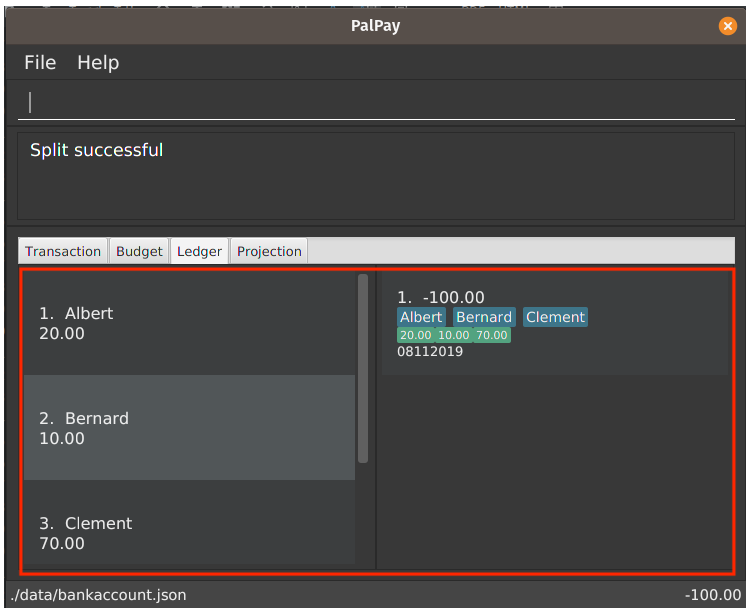

Figure 5. Display of Correctly Assigned Amounts

Since the number of shares is equal to the number of people listed, you are not included in the splitting of the bill.

3.2. Receiving Money from a Friend : `receive`

Receives money from a friend
Format: `receive $/AMOUNT n/NAME1 [d/DATE] [a/DESCRIPTION]`

3.2.1. Example Usage

- `receive $/20 n/Albert`

```
Transfers $20 from Albert to user. If Albert is no longer owe or is owed money, he
will be removed from the Ledger.
```

1. Enter appropriate command into the command line.

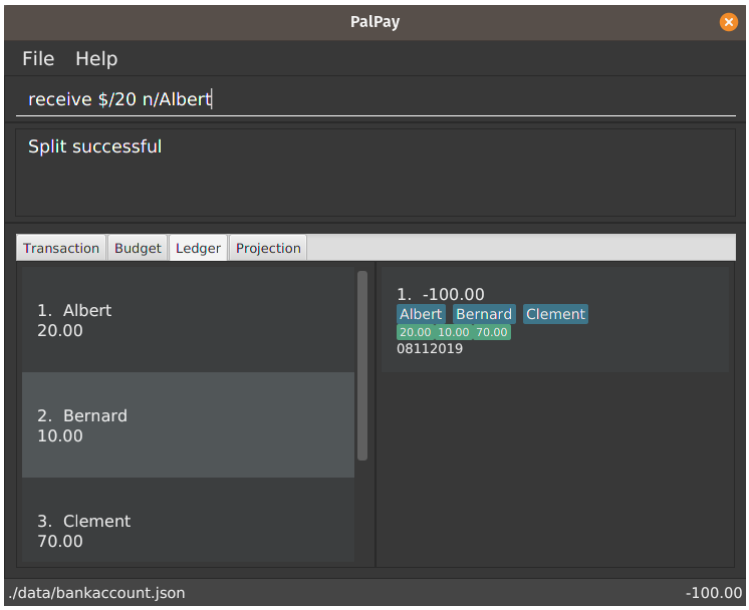

Figure 6. Input for Receiving $20 from Albert

2. Result is displayed accordingly.

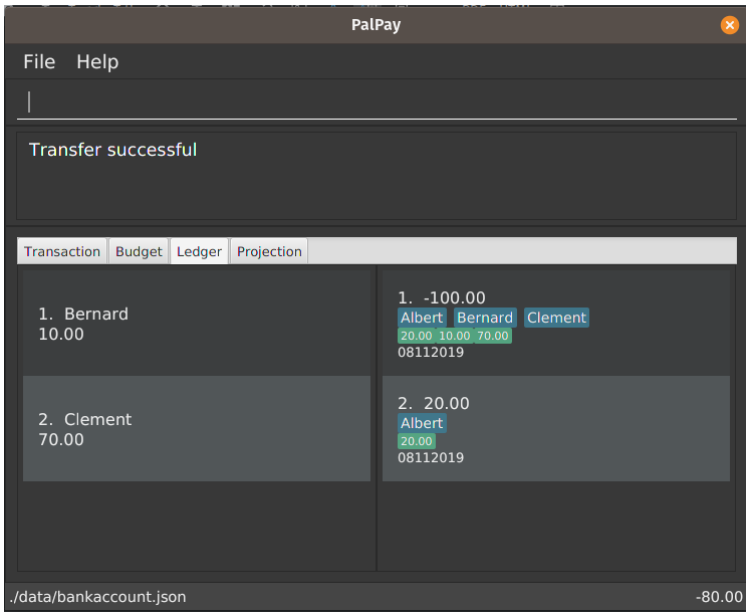

Figure 7. Result of Payment from Albert

Albert is removed from the *Ledger* since he no longer owes any money. *Ledger* balance is also updated accordingly.

4. Contributions to the Developer Guide

Given below are sections I contributed to the Developer Guide. They showcase my ability to write technical documentation and the technical depth of my contributions to the project.

4.1. Split Feature: `split`

This feature allows the user to pay for a certain item or make a transaction on behalf of his friends. Refer to the UserGuide for usage details.

4.1.1. Current Implementation

The `split` command is an abstraction of `LendMoney` class.
Given a list of **shares** and **people**, each person is assigned an **amount** based on the corresponding positional share and the total amount given to `split` command.
A `LendMoney` instance is created for each person and executed.

Below shows how a typical command from the user is executed by the program.

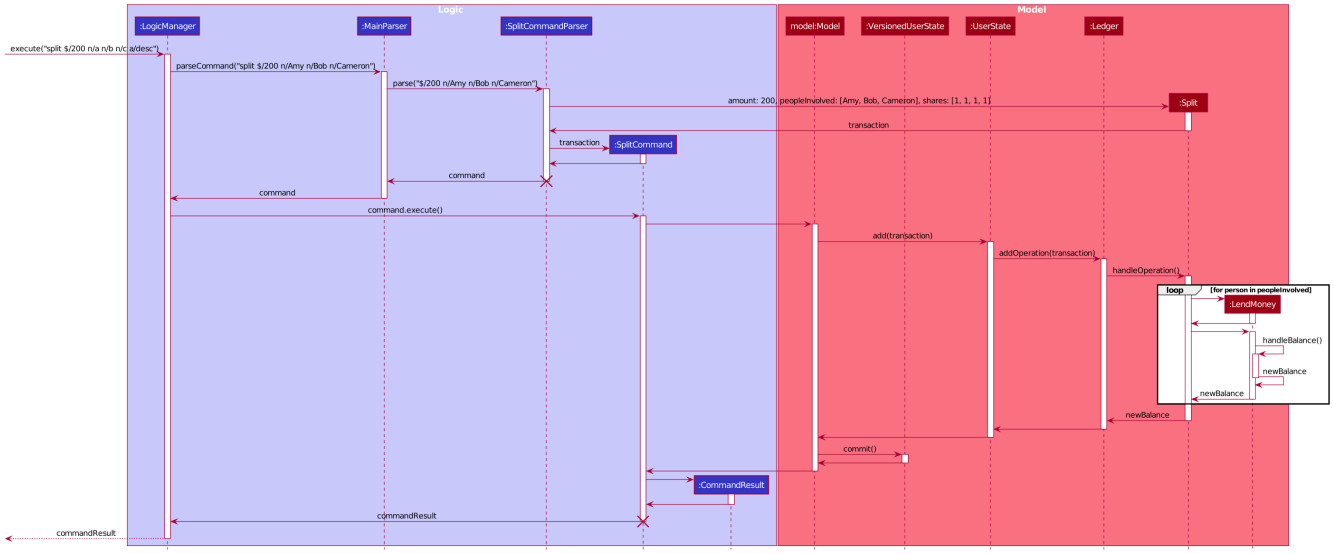

Figure 8. Sequence Diagram for Executing a `SplitCommand`

Step 1: User enters `split $/200 n/a n/b n/c a/desc` into the command line to split **$200** among **a**, **b** and **c**.

Step 2: Upon executing the command, `LogicManager` uses `MainParser#parse()` to parse the input from the user.

Step 3: `MainParser` determines that user input is an instance of a `SplitCommand` and creates a `SplitCommandParser` to further parse the input.

Step 4: `SplitCommandParser` parses the command and checks if fields like amount and names are valid. If all fields are valid, it returns a `SplitCommand`. Else, an error is thrown to the result box and the execution terminates.

Step 5: `LogicManager` uses `SplitCommand#execute()` to update the balances of `Ledger` and people involved in the transaction.

Step 6: `SplitCommand#execute()` calls `Model#add()` to add the user input into the user history. Within

the function call, the actual update of balances is handled in `Ledger#handleOperation()`.

Step 7: `SplitCommand` calls `Model#commitUserState()` after executing to save the latest state of the application.

Step 8: A `CommandResult` is then returned to the `LogicManager` which is then displayed to the user.

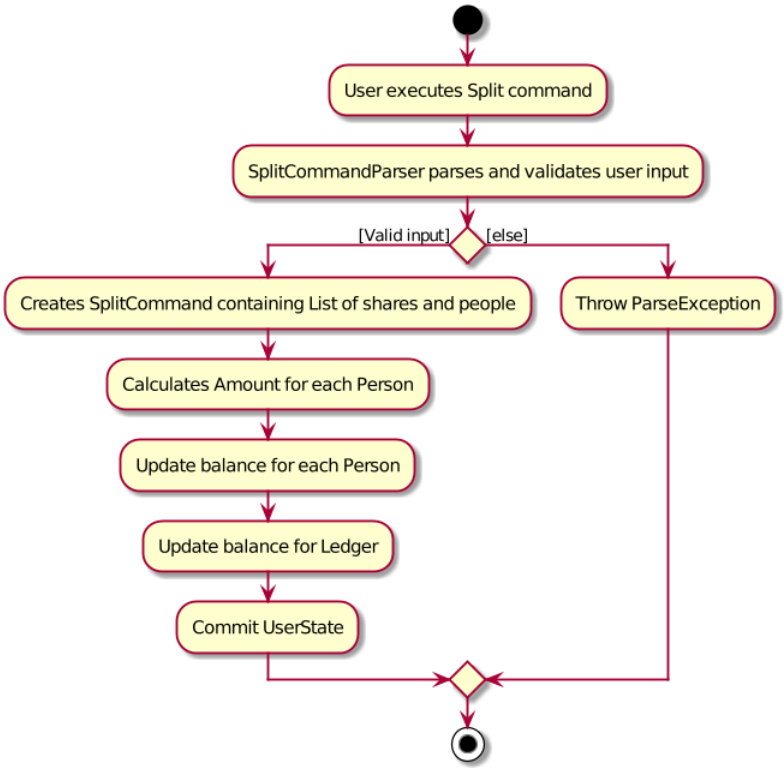

Figure 9. Activity Diagram for Creating a `Split` Object

4.1.2. Design Considerations

Below shows how the classes involved with `Ledger` interact with each other.

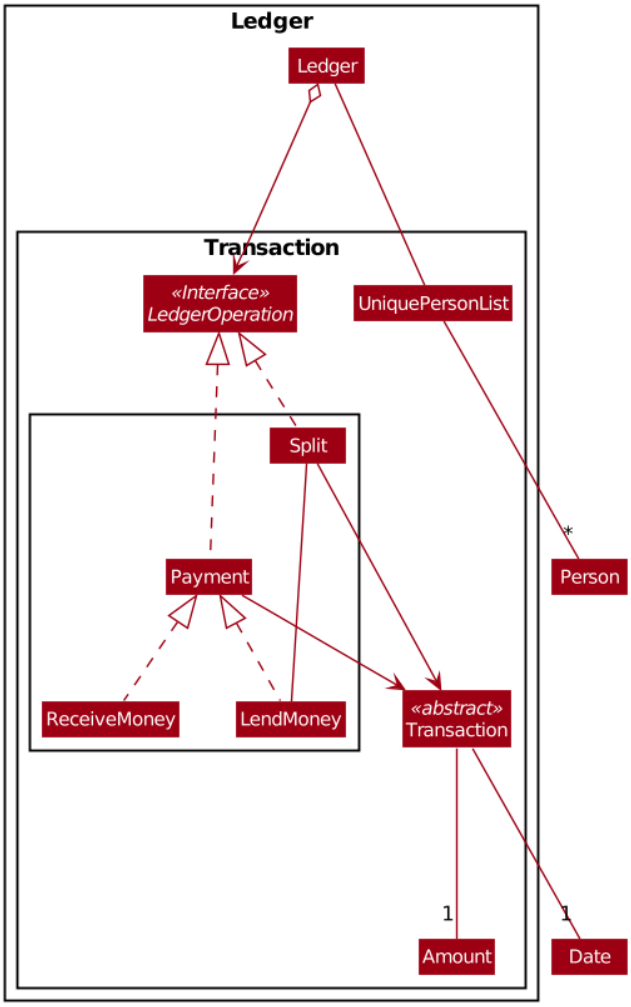

Figure 10. Class Diagram for Operations that Deal with Ledger

Current implementation of `Split` class encourages code reuse by abstracting the delegating the task of rebalancing to another class.
However, this introduces coupling as the behavior of `Split` is now inexplicably tied to `LendMoney`.

4.2. Settle Up Feature: `receive`

This feature allows another person to send money to the user.
The balance in the `Ledger` and the balance of the sender is updated accordingly.

4.2.1. Current Implementation

The `receive` command creates `ReceiveMoney` class that handles the transfer of fund from another person to the user.

How `receive` relates to the rest of the `Ledger` classes can be inferred from the class diagram above.

In the `handleBalance` method of `ReceiveMoney`, it will find the correct person in the `Ledger` by name, or create a new `Person` with given **name** if it is not already in the `Ledger`.
Balance of the user and the sender is then updated accordingly.

Below is an example usage scenario for how `receive` would behave.

Step 1: User enters `receive $/200 n/a` into the command line to settle up **$200** from **a**.

Step 2: Upon executing the command, `LogicManager` uses `MainParser#parse()` to parse the input from the user.

Step 3: `MainParser` determines that user input is an instance of a `ReceiveCommand` and creates a `ReceiveCommandParser` to further parse the input.

Step 4: `ReceiveCommandParser` parses the command and checks if fields like amount and names are valid. If all fields are valid, it returns a `ReceiveCommand`. Else, an error is thrown to the result box and the execution terminates.

Step 5: `LogicManager` uses `ReceiveCommand#execute()` to update the balances of `Ledger` and the person in the transaction.

Step 6: `ReceiveCommand#execute()` calls `Model#add()` to add the user input into the user history. Within the function call, the person in `Ledger` is found and his/her outstanding balance is updated in `Ledger#handleOperation()`.
If he is not already inside the `Ledger`, a new `Person` is created with the same name.

Step 7: `SplitCommand` calls `Model#commitUserState()` after executing to save the latest state of the application.

Step 8: A `CommandResult` is then returned to the `LogicManager` which is then displayed to the user.

Code snippet of `handleBalance` in ReceiveMoney

```
public class ReceiveMoney extends Payment {
    @Override
    public Amount handleBalance(Amount balance, UniquePersonList peopleInLedger) {
        Person target = super.handleTarget(peopleInLedger);
        target.spend(amount);
        return balance.addAmount(amount);
    }
}

public abstract class Payment extends Transaction implements LedgerOperations {
    protected Person handleTarget(UniquePersonList peopleInLedger) {
        Person personInvolved = person;
        if (peopleInLedger.contains(person)) {
            personInvolved = peopleInLedger.get(person).get();
        } else {
            peopleInLedger.add(person);
        }
        return personInvolved;
    }
}
```