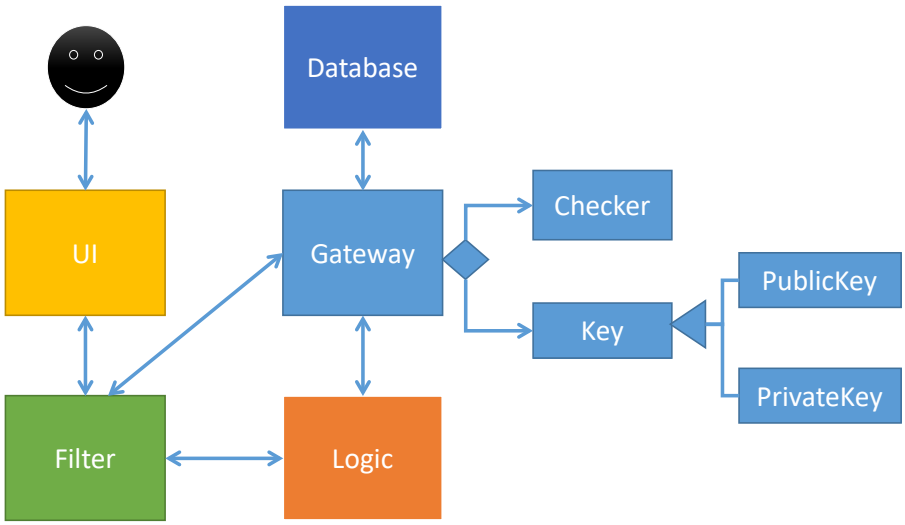
Tutorial 10 – DG Extracts

Contents

What to do? For each of the DG extracts given below, find problems and areas to improve. Refer Tutorial 10 – Task 3 for info on what problems to look for and tips to apply for the DG.

[Ex 1] Architecture diagram



[Ex 2] Class diagram - Tags

3.2. Tag feature

3.2.1. Implementation

The tag mechanism is facilitated by UniqueTagList. It creates a list of Tag, stored internally as an uniqueTagList. Additionally, it implements the following operations:

- `AddTag` - creates a new tag in AlgoBase's uniqueTagList in the algobase history.
- `DeleteTag` - deletes a current tag which have already in the uniqueTagList.
- `EditTag` - edits the current tag name which have already been in the `uniqueTagList`.
- `ListTag` - shows the tags in the uniqueTagList in the algobase GUI for users.

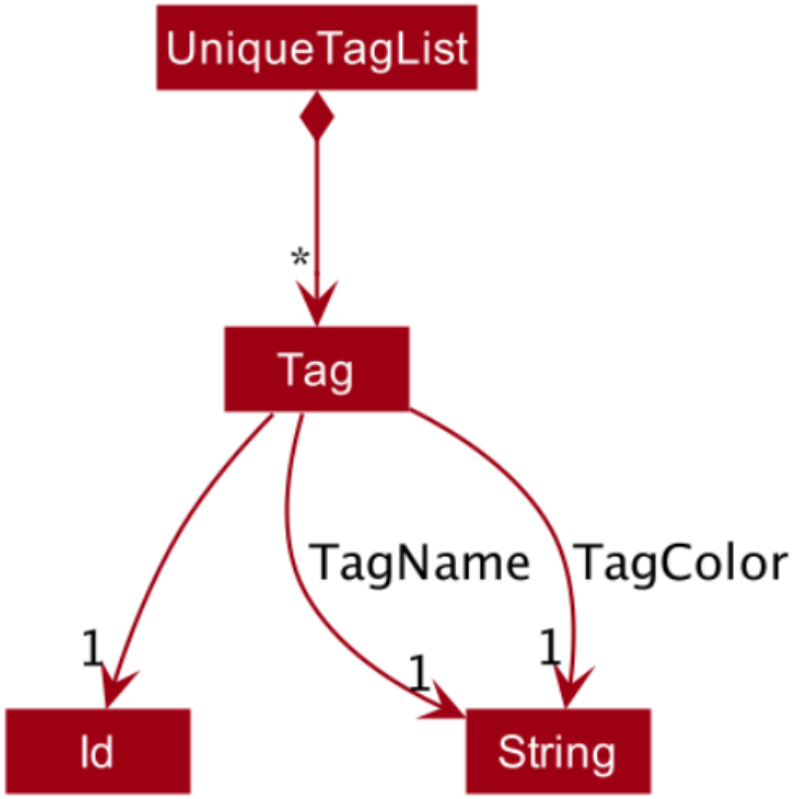


Figure 25. Class Diagram for Tag

[Ex 3] Class diagram - FindPlanDescriptor

Step 5. The user then decides to execute the command `findplan start/2019-03-01 end/2019-03-31` to find out what plans he has in March. The `findplan` command constructs a `FindPlanDescriptor`, and then executes `Model#getFilteredPlanList()` and `Model#updateFilteredPlanList(FindPlanDescriptor)`. A list of plans in AlgoBase that has overlapping time range with the specified starting date and end date will be displayed on the plan list panel.

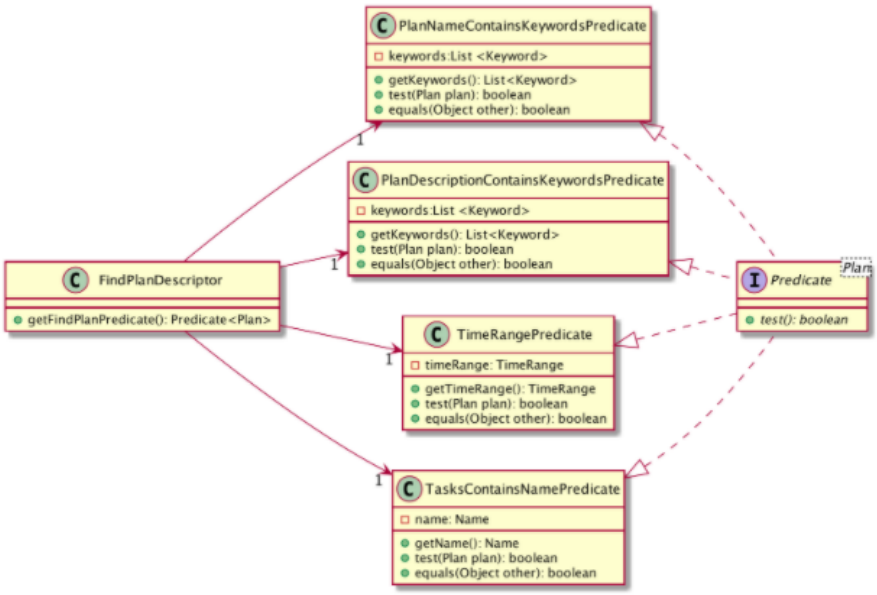


Figure 51. Class Diagram for `FindPlanDescriptor`

[Ex 4] Class diagram - Occurrence

4.5. Cloning transactions

The **clone** feature creates one or more duplicates of a specified `Transaction` and adds them to the end of the existing transactions list.

4.5.1. Implementation

An `Index` and `Occurrence` are obtained from their representation in user input. The `Index` specifies which transaction to clone, while the `Occurrence` informs THRIFT how many clones of the transaction should be created (`Occurrence#numOccurrences`) and the time period between them (`Occurrence#frequency`).

Here is a Class Diagram for the implementation of `Occurrence`:

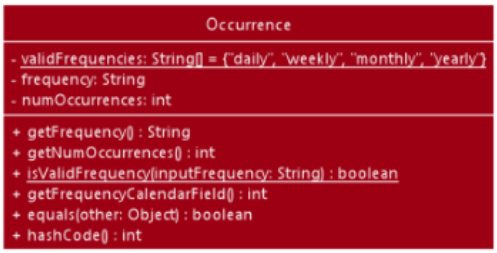


Figure 17. Implementation of `Occurrence` *class*

[Ex 5] Class diagram – Model component

3.4. Model Component

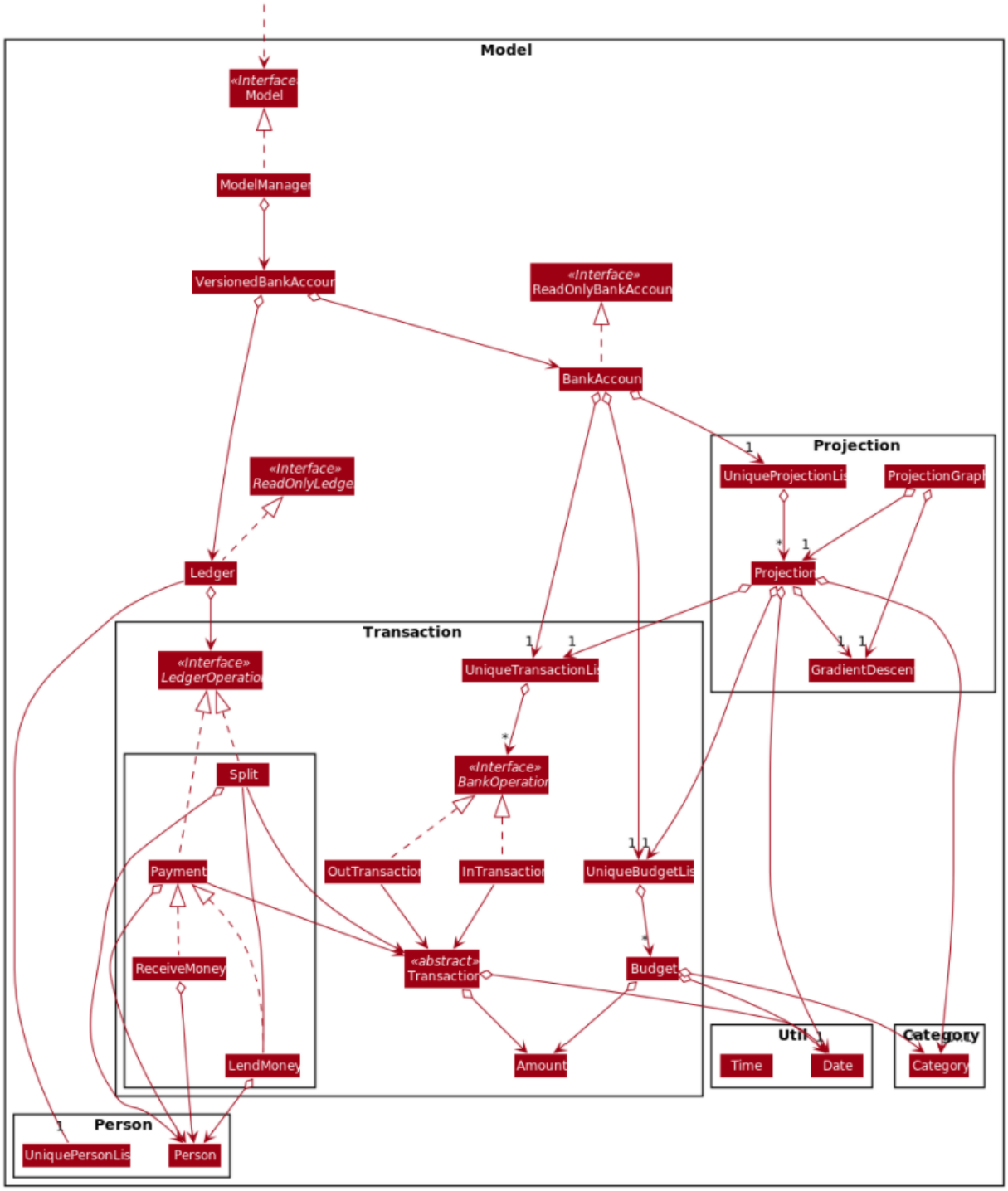


Figure 8. Structure of the Model Component

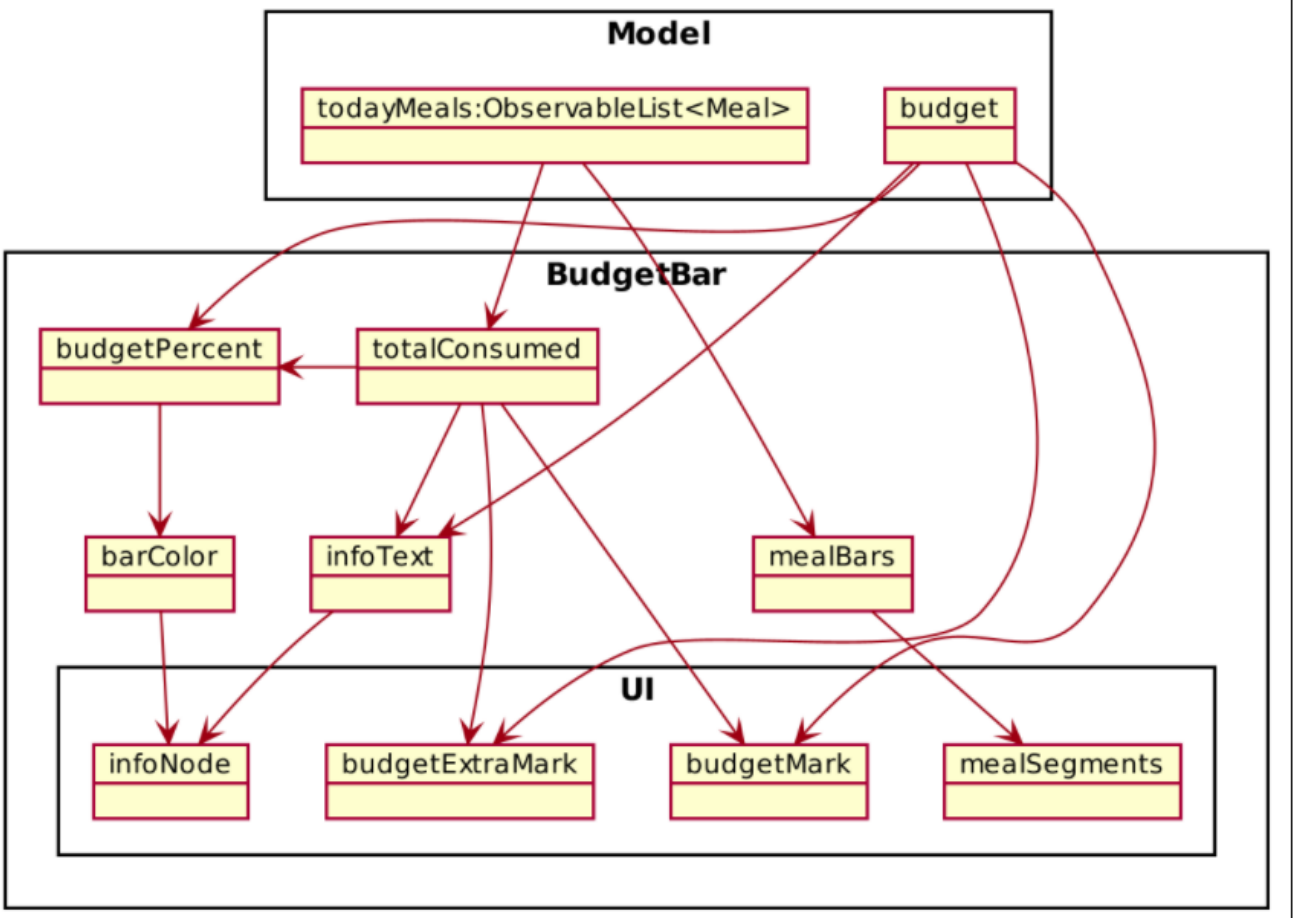
API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user's preferences.

[Ex 6] Object diagram – reactive update dependencies

The following object diagram shows the reactive update dependencies.



[Ex 7] Object diagram for the update command

The following Object Diagram illustrates objects involved in the execution of **update** command:

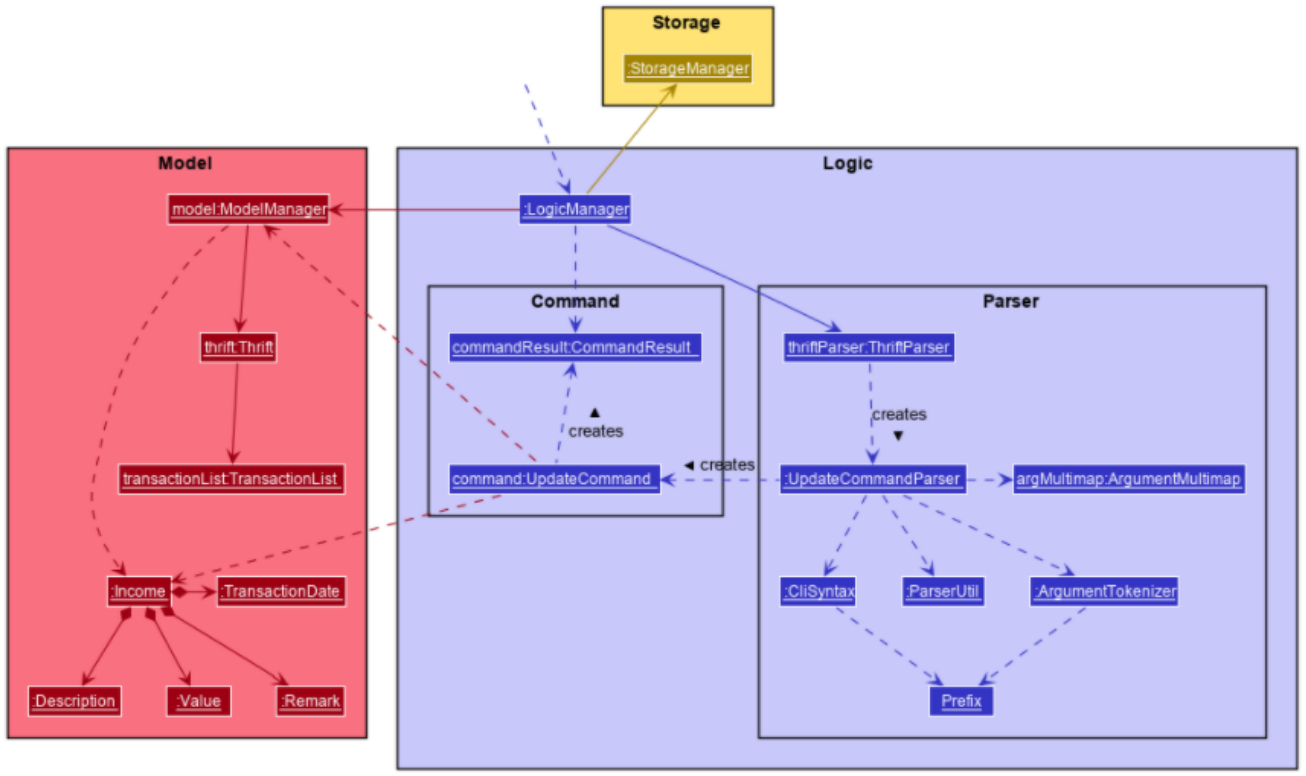
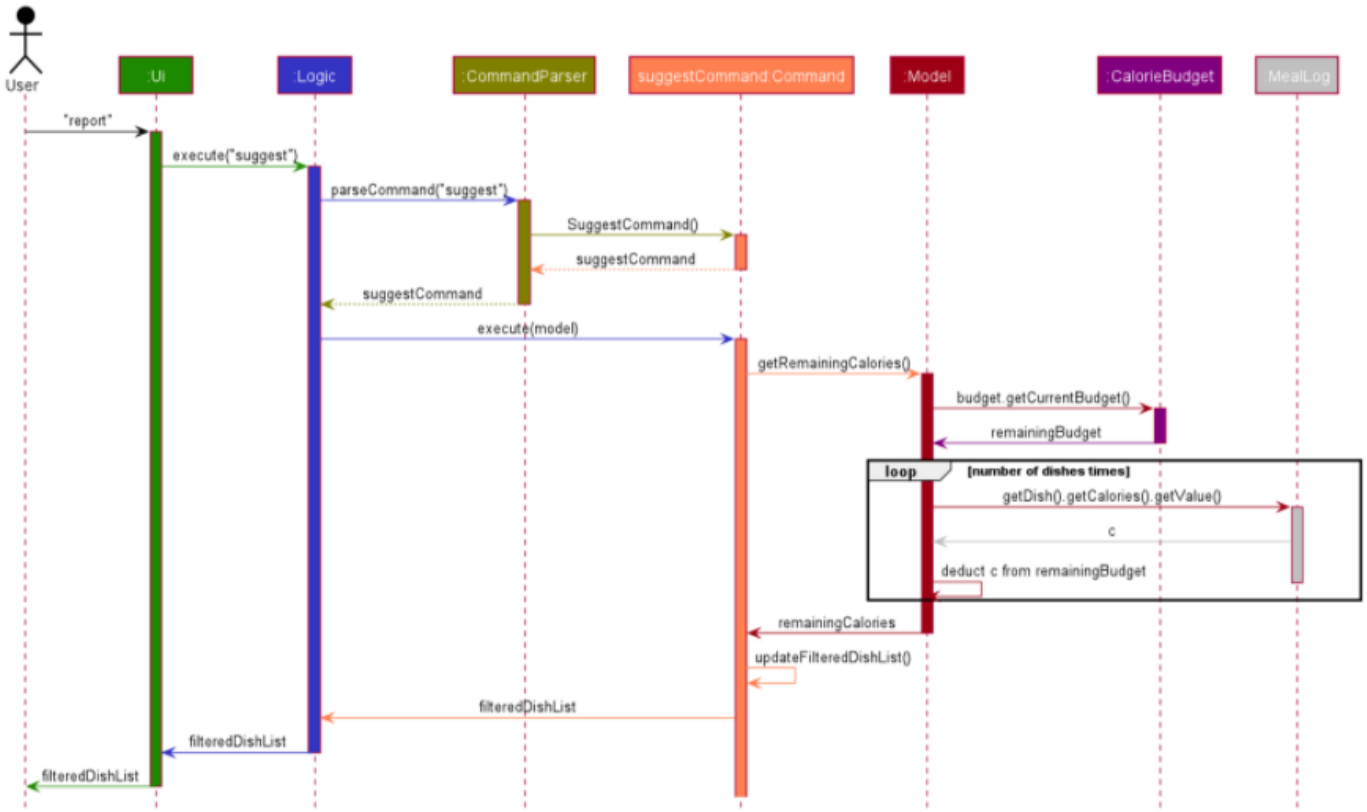


Figure 15. Existing objects when executing ***update*** *on an* `Income`

[Ex 8] Sequence diagram – suggest operation

The following sequence diagram shows how the suggest operation works:



[Ex 9] Sequence diagram – split command

4.3.1. 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.

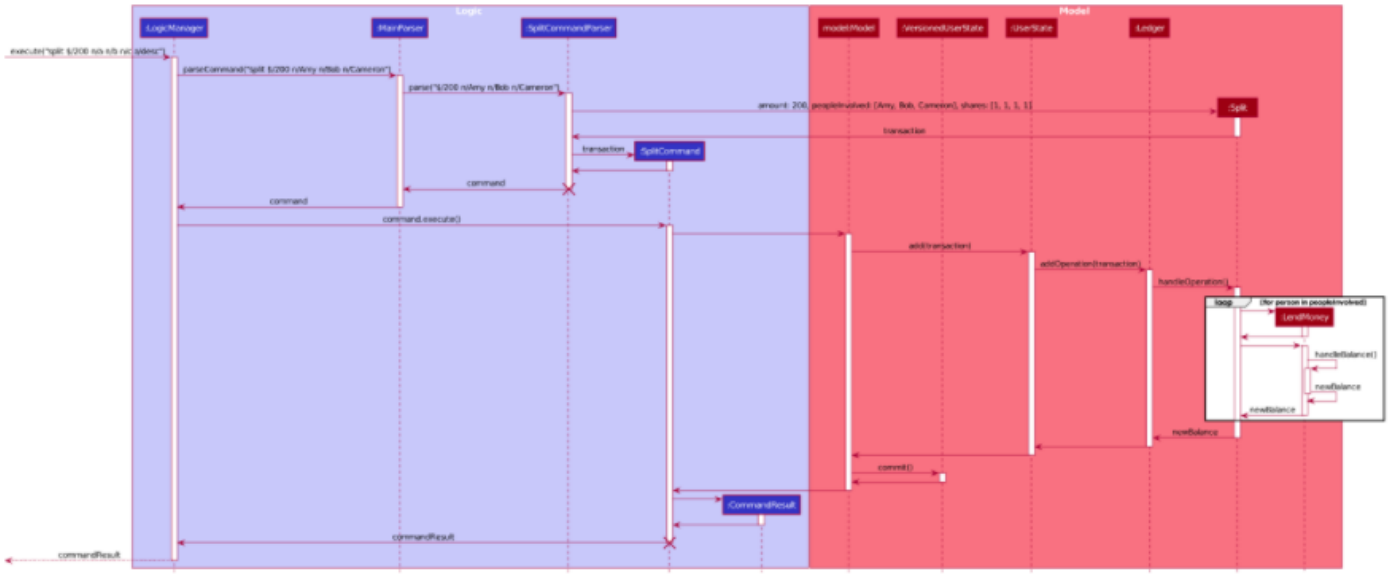


Figure 18. Sequence Diagram for Executing a `SplitCommand`

[Ex 10] Activity diagram – UI context switching

3.6.3. UI context switching

After executing a command successfully, the `MainWindow` receives a `CommandResult`, which it uses to determine if any additional actions need to be performed.

If the `CommandResult` contains a `Context`, then `MainWindow` switches the currently displayed panel out for the appropriate panel specified by the `ContextType`. The following activity diagram encapsulates the additional actions that may be performed as a result of `MainWindow` parsing the `CommandResult`.

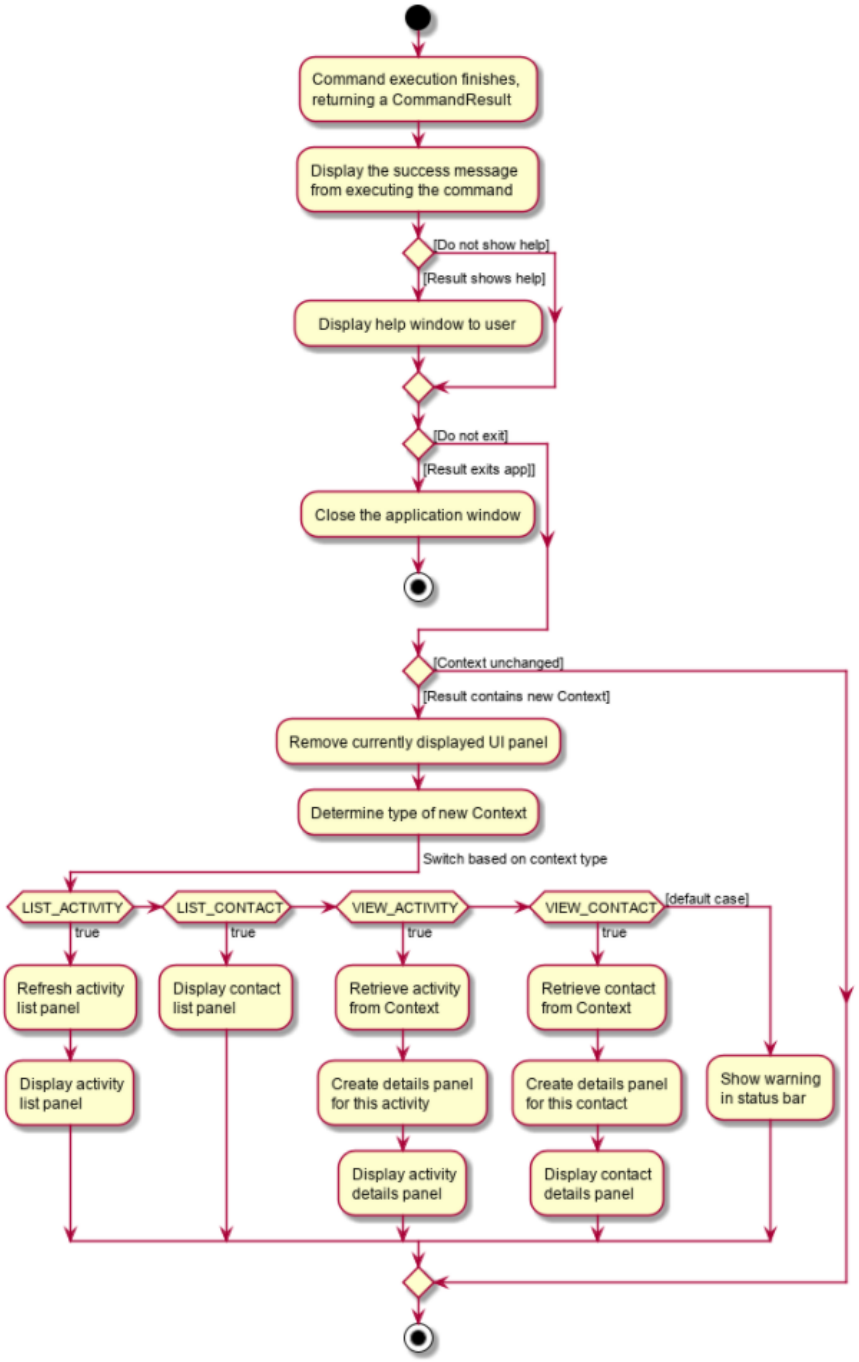


Figure 16. Activity diagram for UI after successful command execution

[Ex 11] Activity diagram – open new tab

The following Activity diagram illustrates the series of actions that occur when the user opens a new tab:

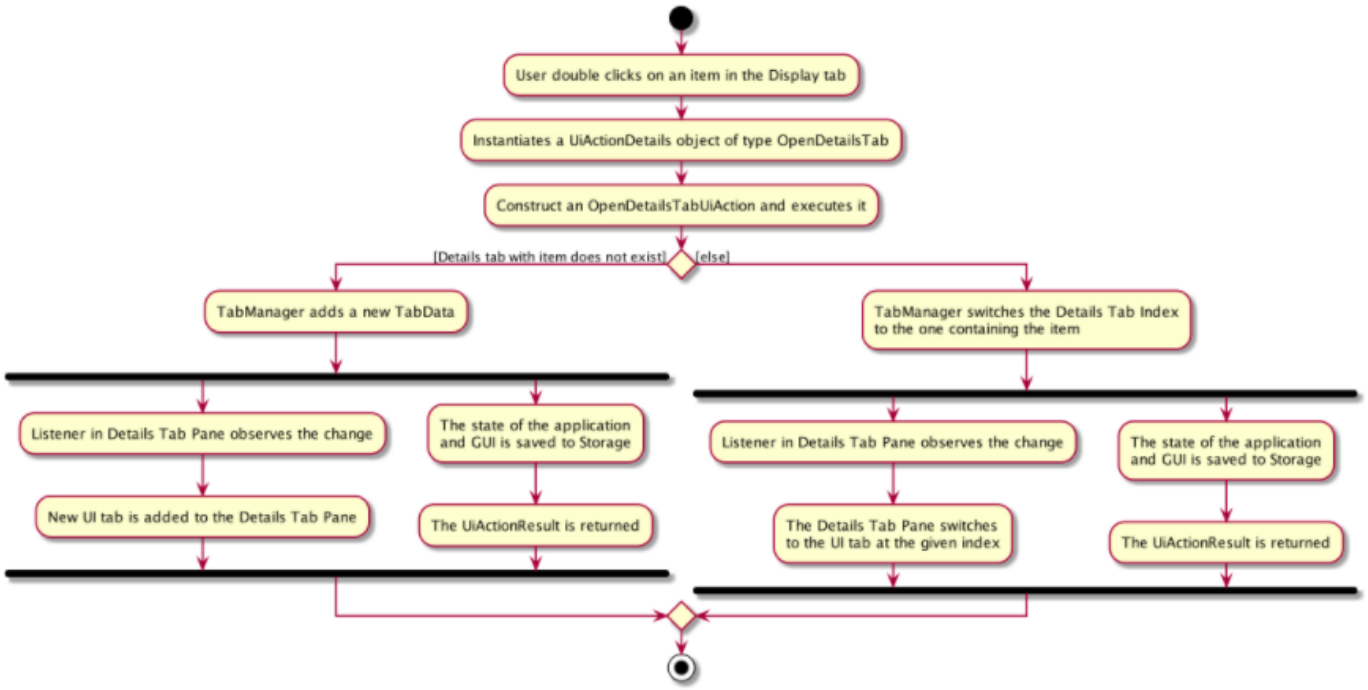


Figure 40. Activity Diagram for Opening a new Tab from the GUI

[Ex 12] Activity diagram – unknown command

Step 7 : `CreateShortCutCommmand` would then return a `CommandResult` to the `LogicManager` which would then be returned back to the user.

The following diagrams summarises what happens when a user executes an unknown command:

Figure 2.4.1 is the activity diagram when a user inputs an unknown command

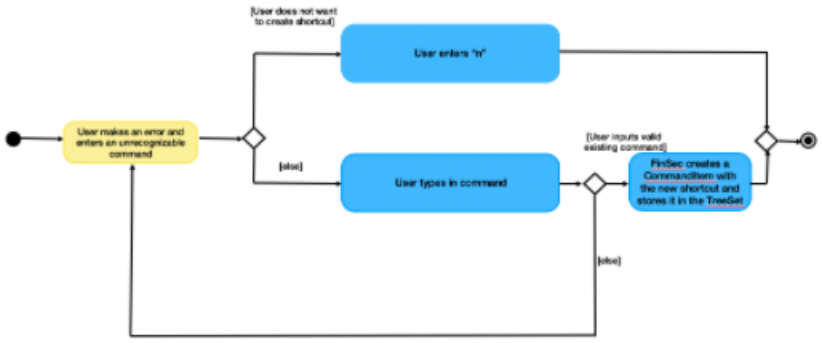


Figure 2.4.1: ActivityDiagram when a user inputs an unknown command

Figure 2.4.2 shows the UML diagram of the flow of logic when a user creates a shortcut to a valid command

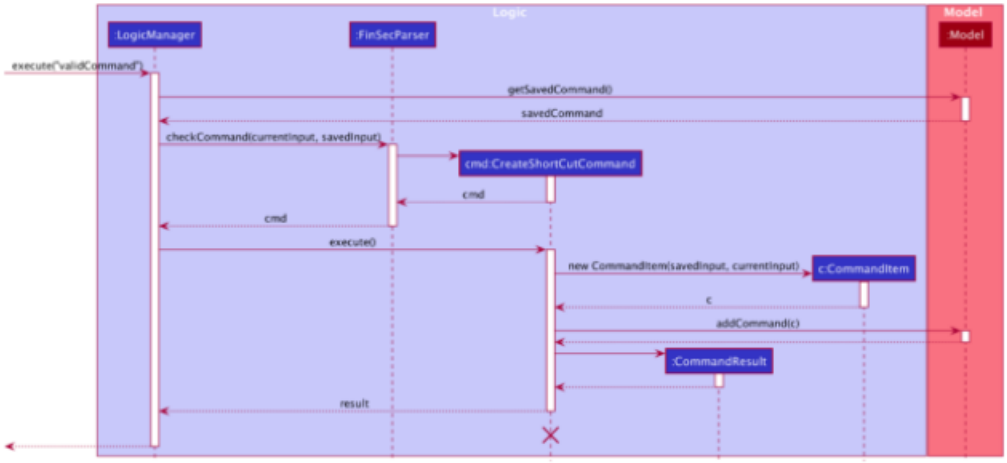


Figure 2.4.2: UML diagram when a user creates a shortcut

[Ex 13] Code examples - sorting

5) The comparators shown below are examples of the various lists are sorted.

- `sortFilteredClaimListByName` is implemented with the help of a comparator that compares the descriptions of each claim with `claim.getDescription()` method. The code snippet below illustrates the comparator.

```
class ClaimNameComparator implements Comparator<Claim> {
    @Override
    public int compare(Claim claim1, Claim claim2) {
        return claim1.getDescription().toString().toUpperCase()
                .compareTo(claim2.getDescription().toString().toUpperCase());
    }
}
```

- `sortFilteredIncomeListByDate` is implemented with the help of a comparator that compares the dates of each income with `income.getDate().getLocalDate()` method. The code snippet below illustrates the comparator.

```
class IncomeDateComparator implements Comparator<Income> {
    @Override
    public int compare(Income income1, Income income2) {
        return income1.getDate().getLocalDate()
                .compareTo(income2.getDate().getLocalDate());
    }
}
```

- `sortFilteredClaimListByStatus` is implemented with the help of a comparator that compares the statuses of each claim. The order is as such: Pending, Approved, Rejected. There are 9 cases of comparison between 2 claims. The code snippet below illustrates the comparator.

```
class ClaimStatusComparator implements Comparator<Claim> {
    @Override
    public int compare(Claim claim1, Claim claim2) {
        if (claim1.getStatus().equals(Status.PENDING) && claim2.getStatus().equals(Status.APPROVED)) {
            return -1;
        } else if (claim1.getStatus().equals(Status.PENDING) && claim2.getStatus().equals(Status.PENDING)) {
            return 0;
        } else if (claim1.getStatus().equals(Status.PENDING) && claim2.getStatus().equals(Status.REJECTED)) {
            return -1;
        } else if (claim1.getStatus().equals(Status.APPROVED) && claim2.getStatus().equals(Status.REJECTED)) {
            return -1;
        } else if (claim1.getStatus().equals(Status.APPROVED) && claim2.getStatus().equals(Status.APPROVED)) {
            return 0;
        } else if (claim1.getStatus().equals(Status.APPROVED) && claim2.getStatus().equals(Status.PENDING)) {
            return 1;
        } else if (claim1.getStatus().equals(Status.REJECTED) && claim2.getStatus().equals(Status.PENDING)) {
            return 1;
        } else if (claim1.getStatus().equals(Status.REJECTED) && claim2.getStatus().equals(Status.REJECTED)) {
            return 0;
        } else if (claim1.getStatus().equals(Status.REJECTED) && claim2.getStatus().equals(Status.APPROVED)) {
            return 1;
        } else {
            return 0;
        }
    }
}
```

[Ex 14] Use cases – add/delete/edit tag

Use Case 7: Add Tag

MSS

1. User requests to add a tag.
2. AlgoBase creates the tag with taq name and tag color.
3. AlgoBase displays the tag list.

   Use case ends.

Extensions

2a. AlgoBase detects that tag name or tag color has an invalid format.

2a1. AlgoBase informs user that the form of new tag is invalid.

Use case ends.

Use Case 8: Delete Tag

MSS

1. User requests to delete a tag.
2. AlgoBase deletes the tag in tag list.
3. AlgoBase deletes the tag in every problems.
4. AlgoBase displays the tag list.

   Use case ends.

Extensions

2a. AlgoBase detects that the index of tag in not valid.

2a1. AlgoBase informs user that the index of tag is invalid.

Use case ends.

Use Case 9: Edit Tag

MSS

1. User requests to edit a tag.
2. AlgoBase edits the tag with taq name and tag color.
3. AlgoBase displays the tag list.

   Use case ends.

Extensions

2a. AlgoBase detects that tag name or tag color has an invalid format.

2a1. AlgoBase informs user that the form of new tag is invalid.

Use case ends.