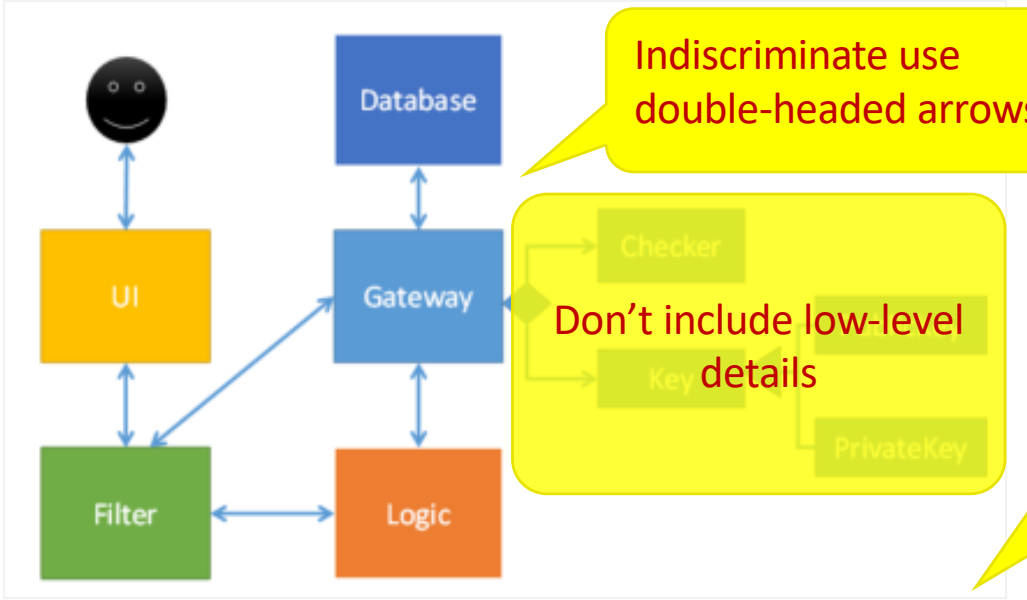
[Ex 1]

Meaning of arrows not clear

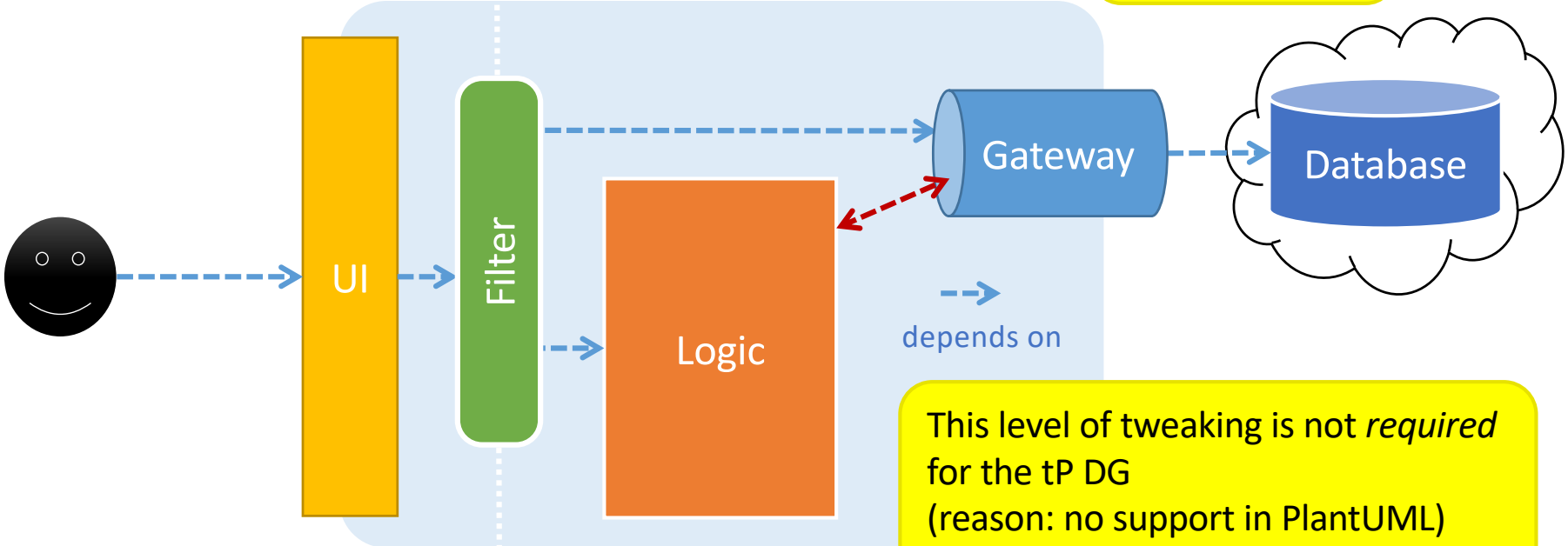


Indiscriminate use double-headed arrows

Don't include low-level details

More meaningful use of,
- arrow direction
- layout
- symbols
- size

Easier to understand and remember



This level of tweaking is not *required* for the tP DG
(reason: no support in PlantUML)

[Ex 2]

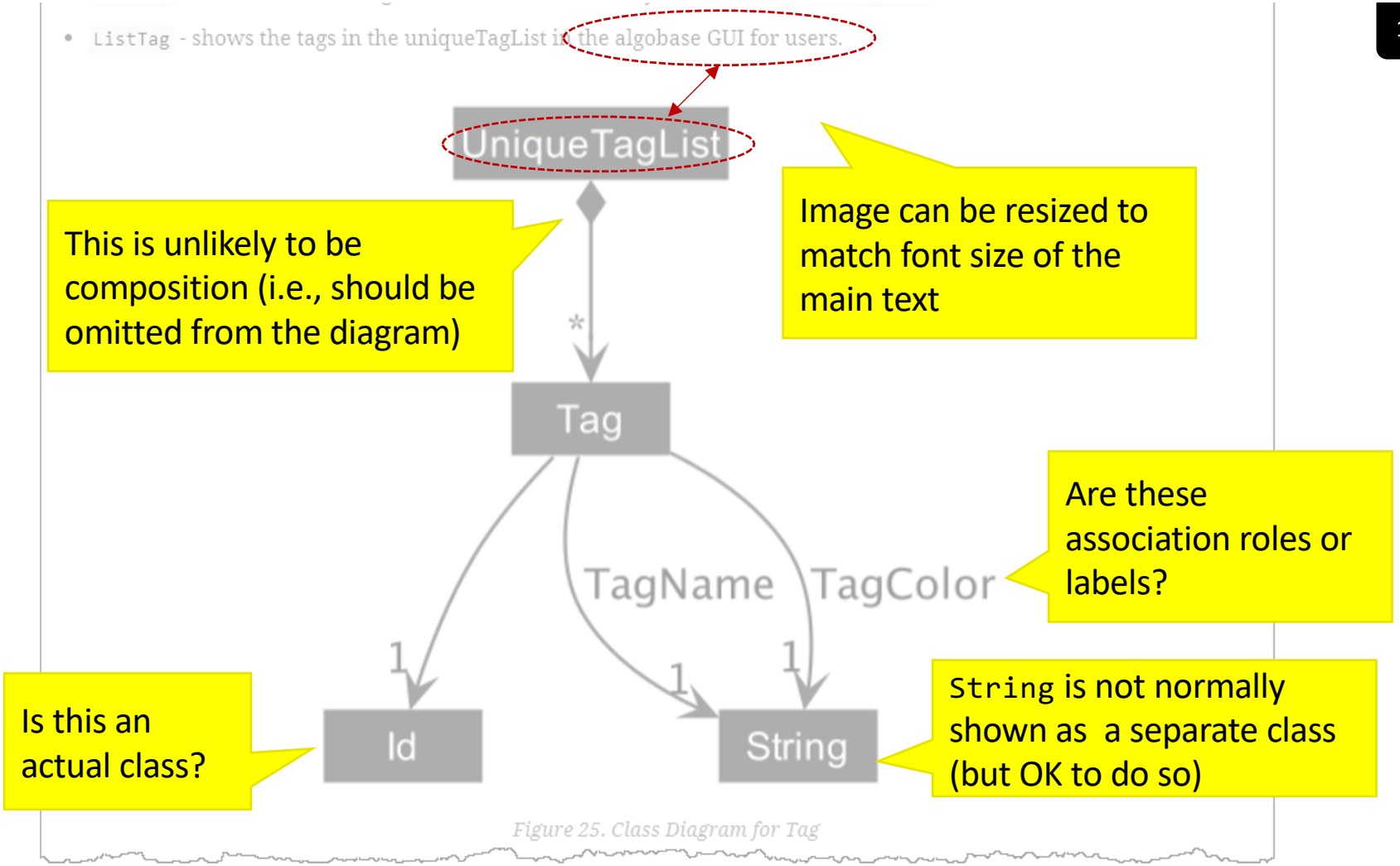


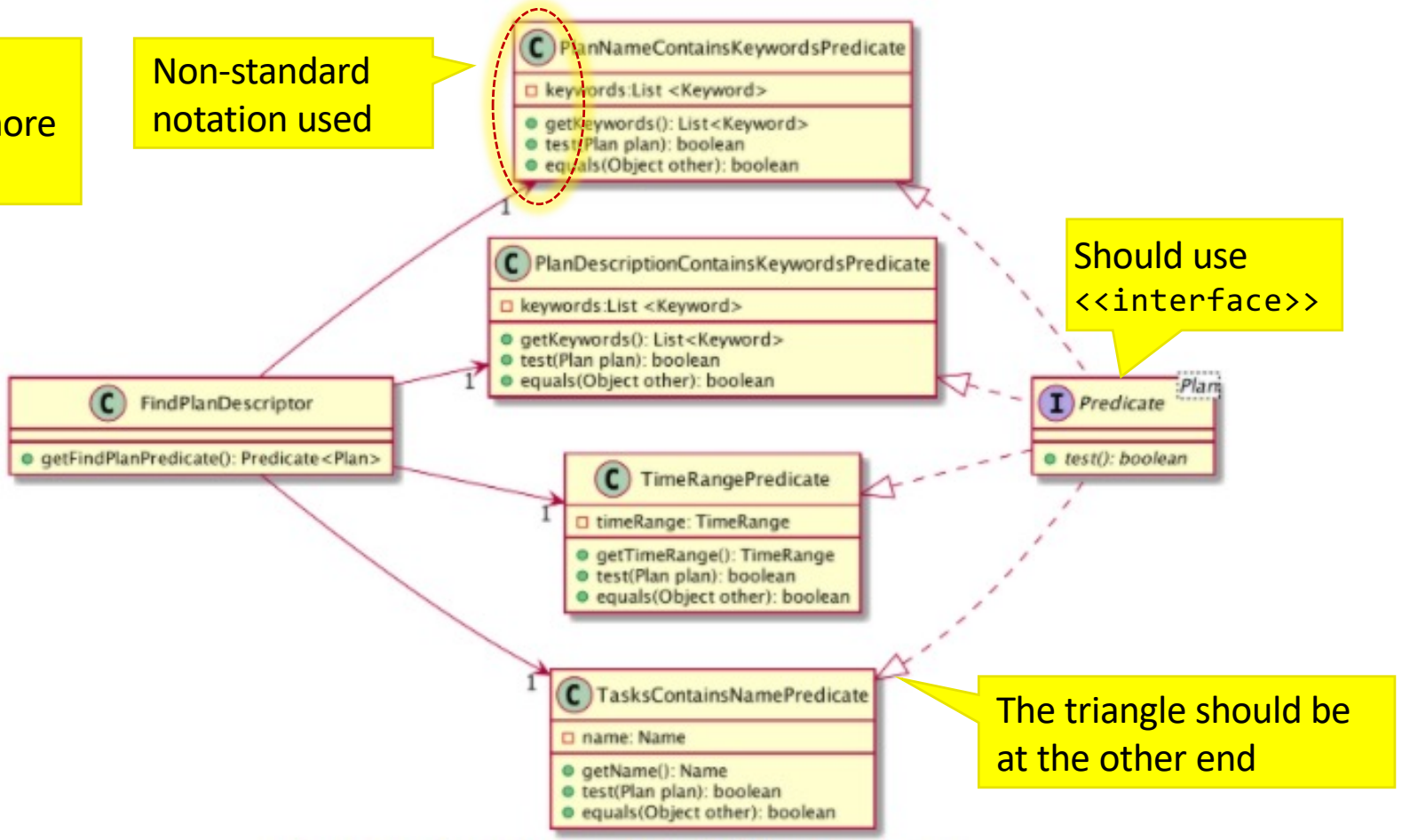
Figure 25. Class Diagram for Tag

[Ex 3]

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.

A sequence diagram is more suited?

Non-standard notation used



Should use `<<interface>>`

The triangle should be at the other end

Figure 51. Class Diagram for FindPlanDescriptor

[Ex 4]

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`:

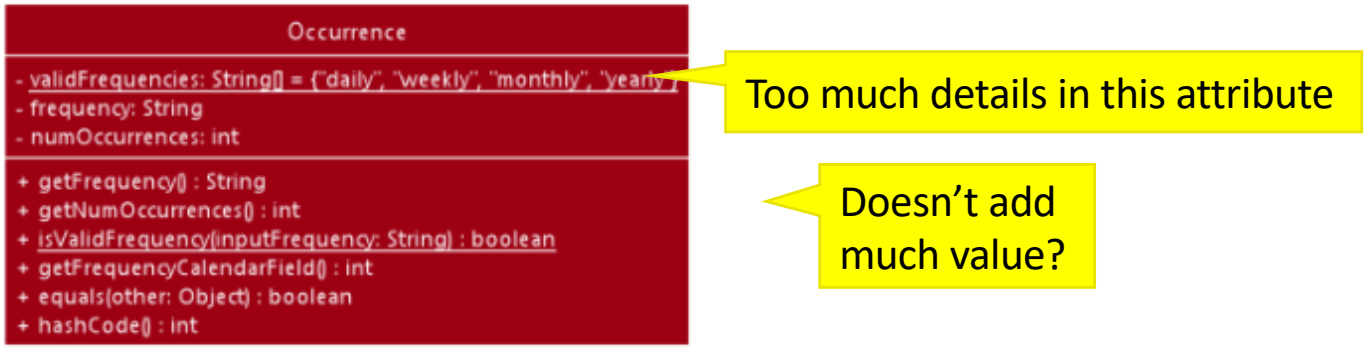


Figure 17. Implementation of `Occurrence` *class*

[Ex 5]

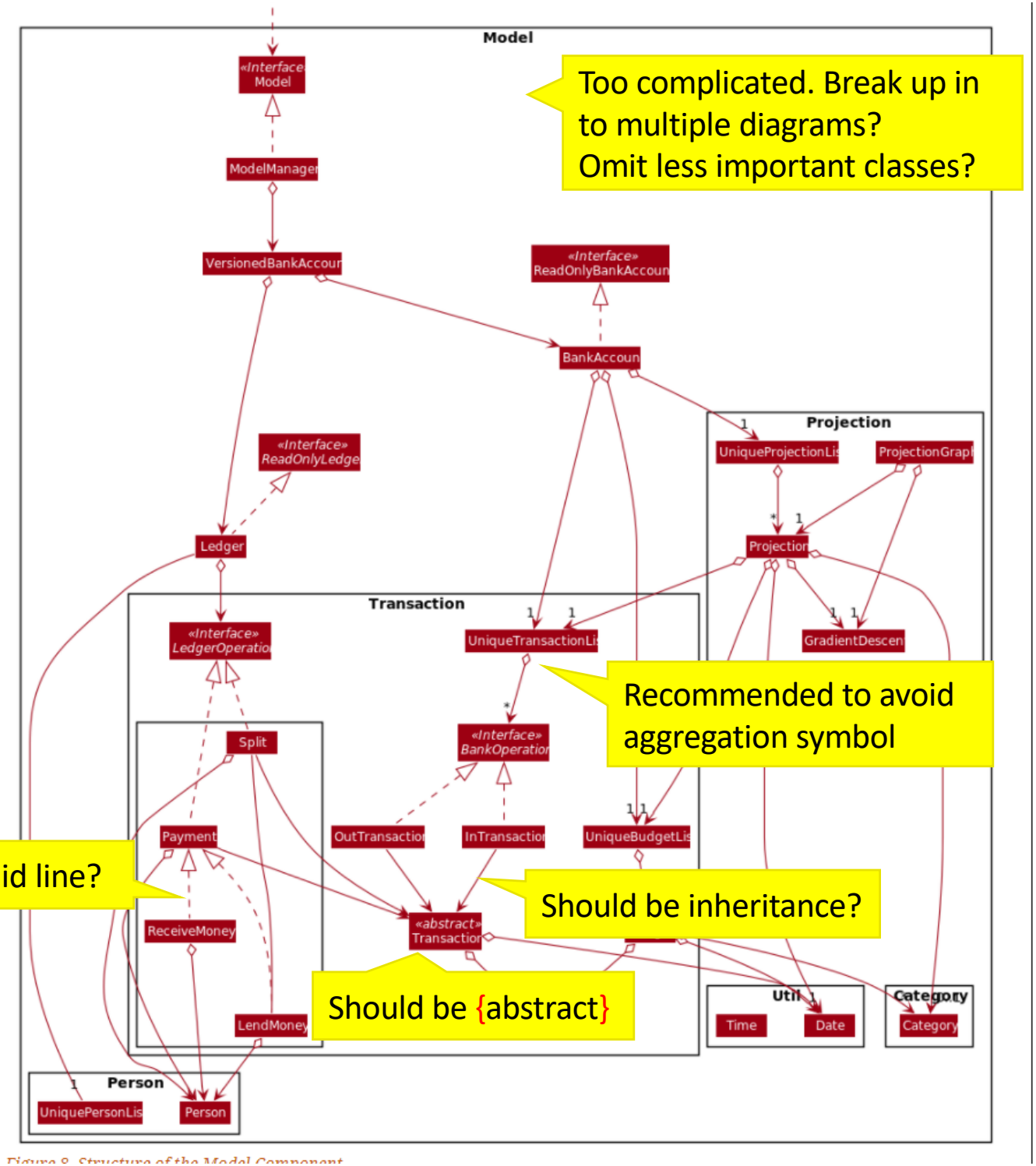
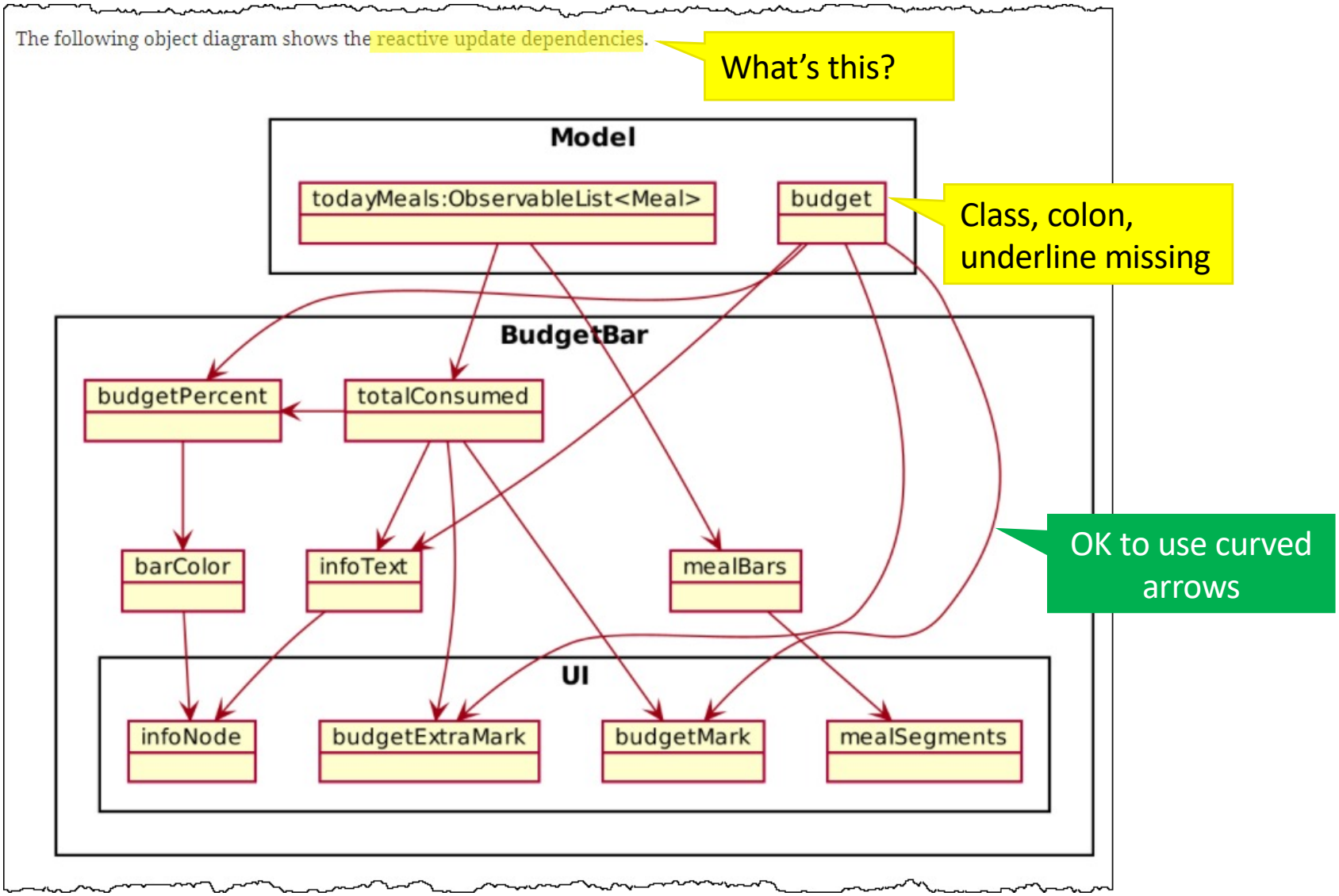


Figure 8. Structure of the Model Component

[Ex 6]


The following object diagram shows the reactive update dependencies.
What's this?
Model
todayMeals:ObservableList<Meal>
budget
Class, colon, underline missing
BudgetBar
budgetPercent
totalConsumed
barColor
infoText
mealBars
OK to use curved arrows
UI
infoNode
budgetExtraMark
budgetMark
mealSegments

[Ex 7]

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

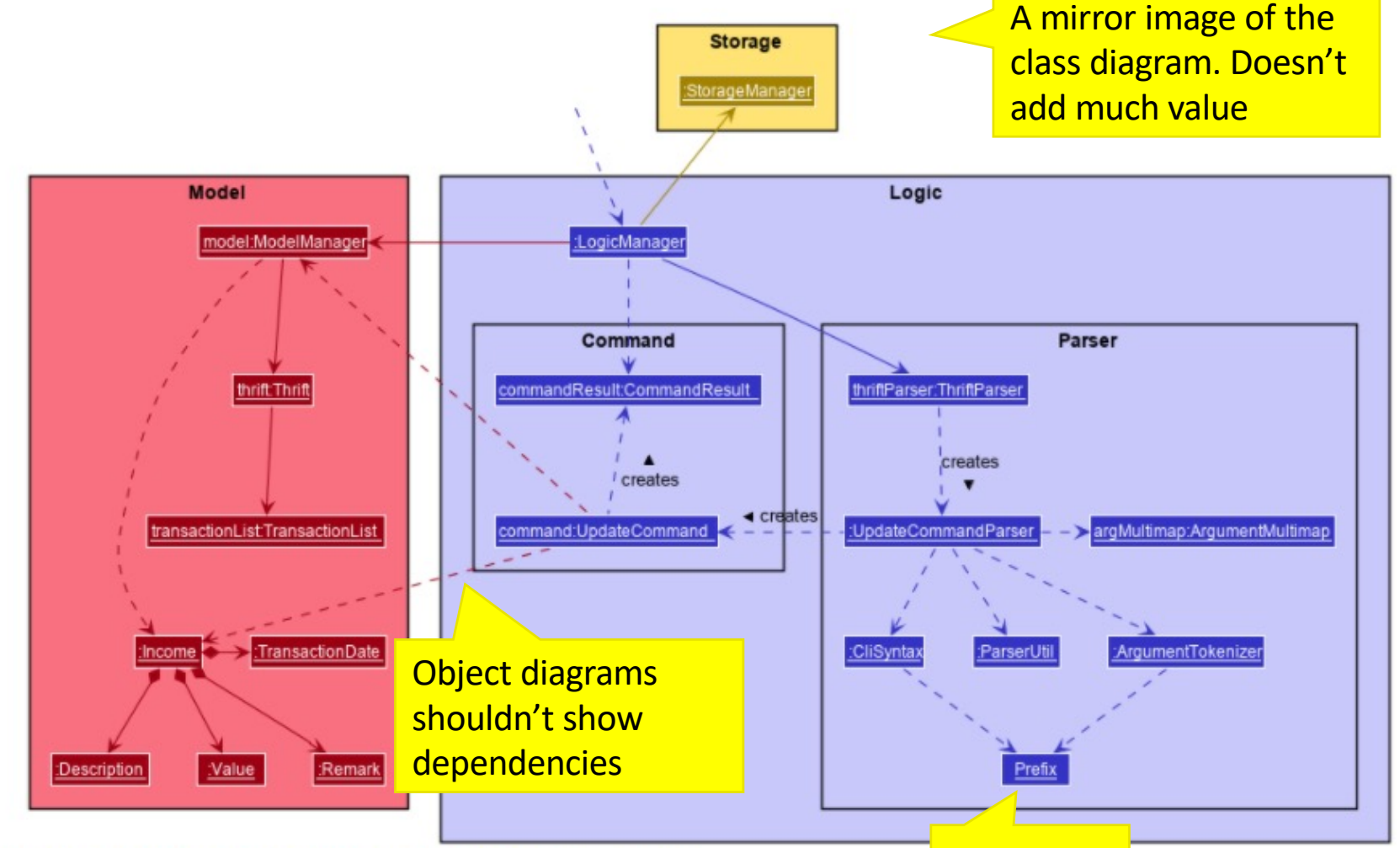
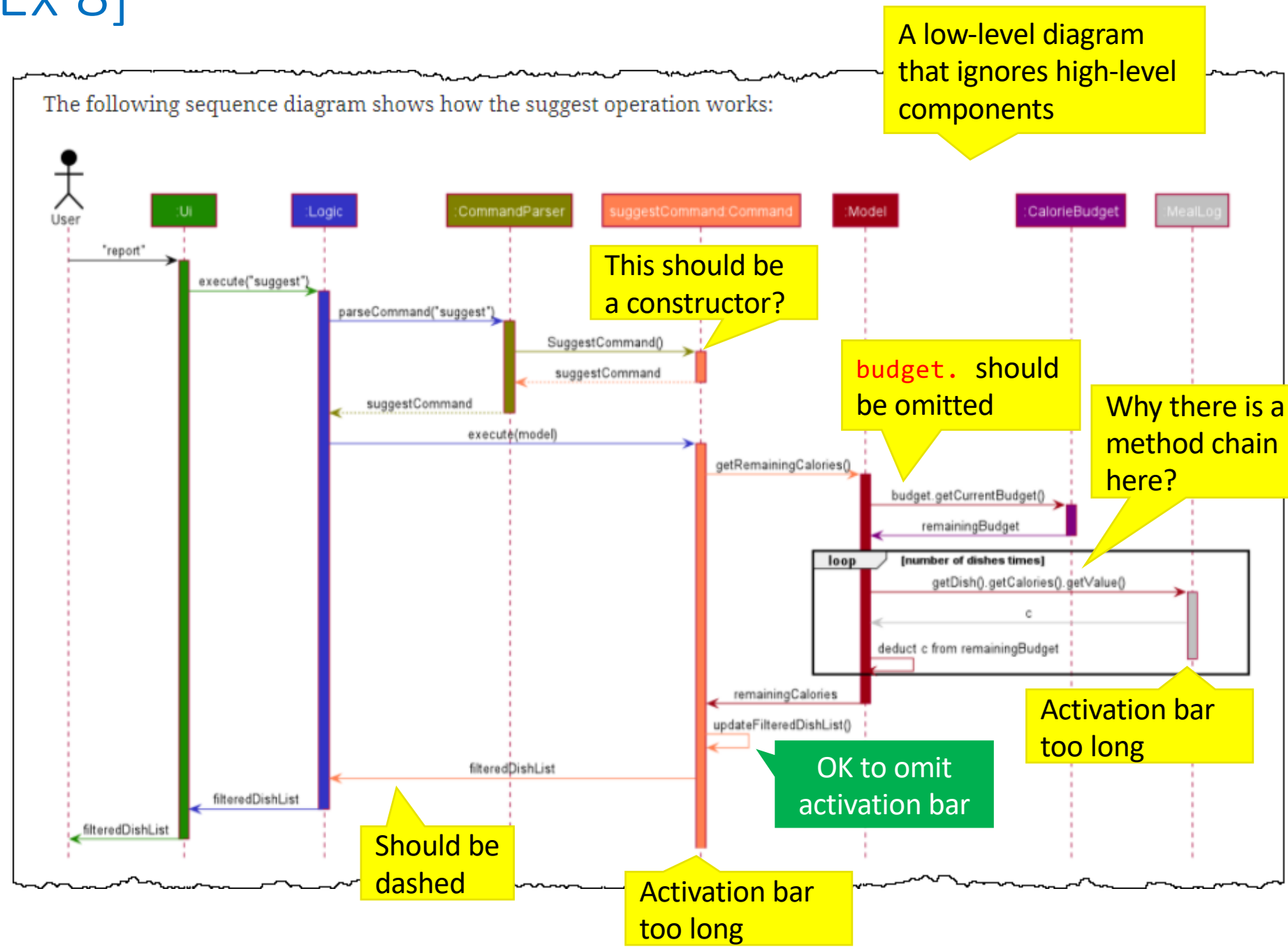
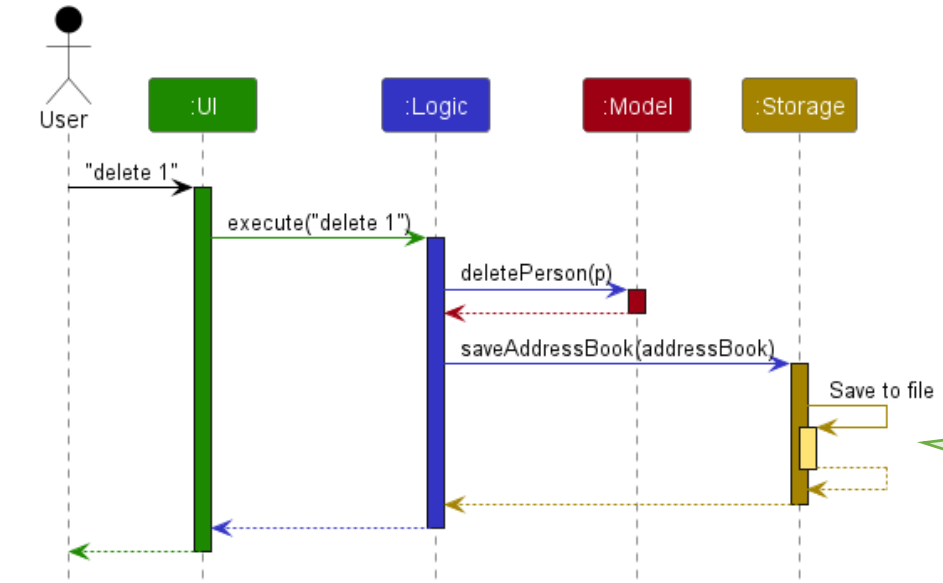


Figure 15. Existing objects when executing ***update*** *on an* `Income`

[Ex 8]


A low-level diagram that ignores high-level components
The following sequence diagram shows how the suggest operation works:
User
:Ui
:Logic
:CommandParser
suggestCommand:Command
:Model
:CalorieBudget
MealLog
"report"
execute("suggest")
parseCommand("suggest")
SuggestCommand()
suggestCommand
suggestCommand
execute(model)
This should be a constructor?
budget. should be omitted
Why there is a method chain here?
getRemainingCalories()
budget.getCurrentBudget()
remainingBudget
loop
[number of dishes times]
getDish().getCalories().getValue()
c
deduct c from remainingBudget
remainingCalories
updateFilteredDishList()
filteredDishList
filteredDishList
filteredDishList
Activation bar too long
OK to omit activation bar
Should be dashed
Activation bar too long

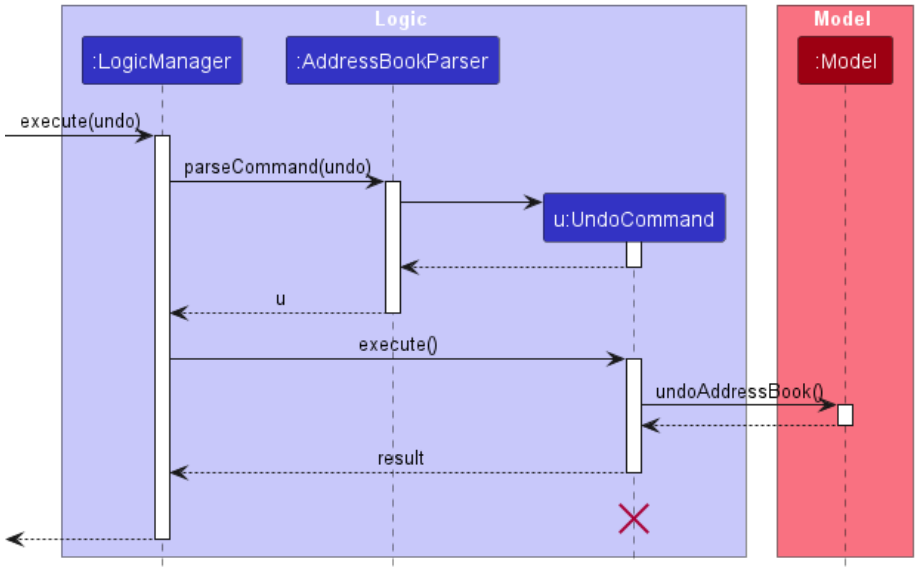
[Ex 9]



Reason: In a big system, the reader should not be forced to know low level detail of more than one component.

Ideally, diagrams that cover multiple components should **show low-level details of at most one component**

Example from AB3 DG:
Shows interactions between high-level components only.



Example from AB3 DG:
Shows low-level interactions within only one high-level component only.

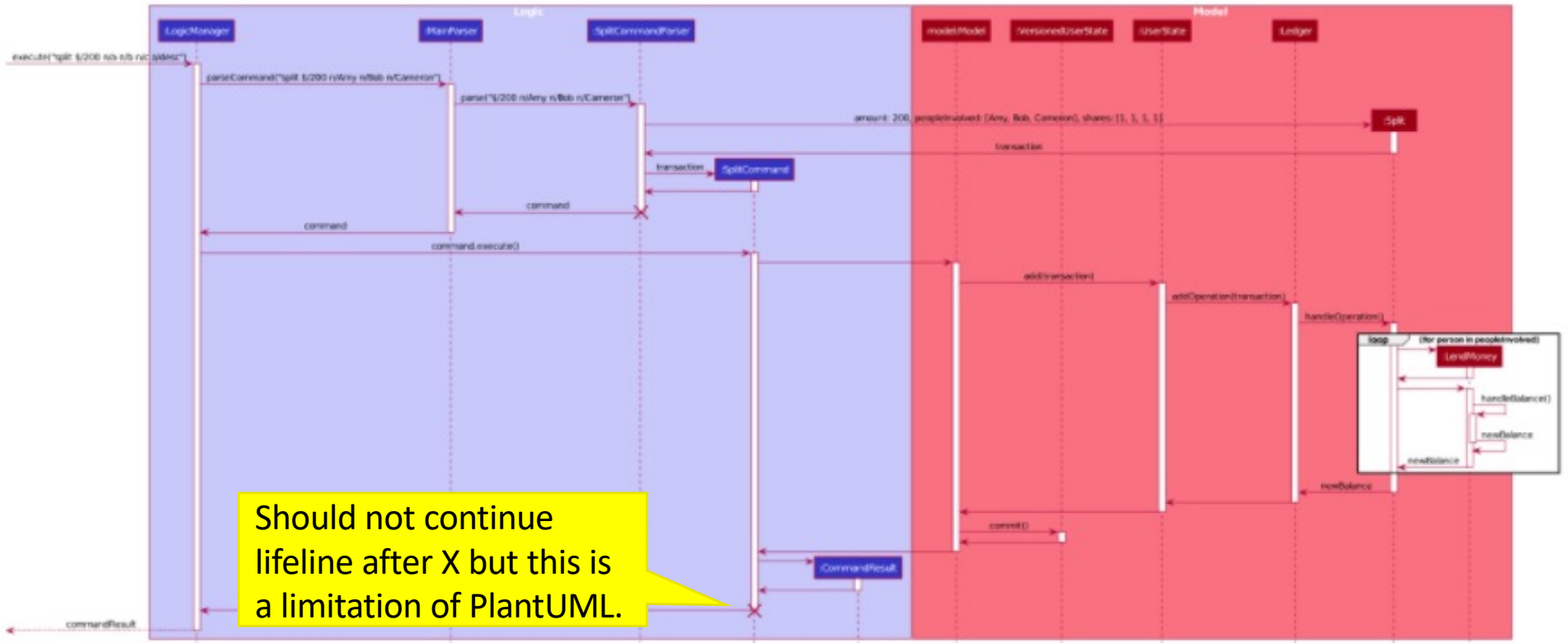
[Ex 9]

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 **amoun**
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.

Too detailed. Don't show low level details of multiple components. Ideally, diagrams that cover multiple components should **show low-level details of at most one component**

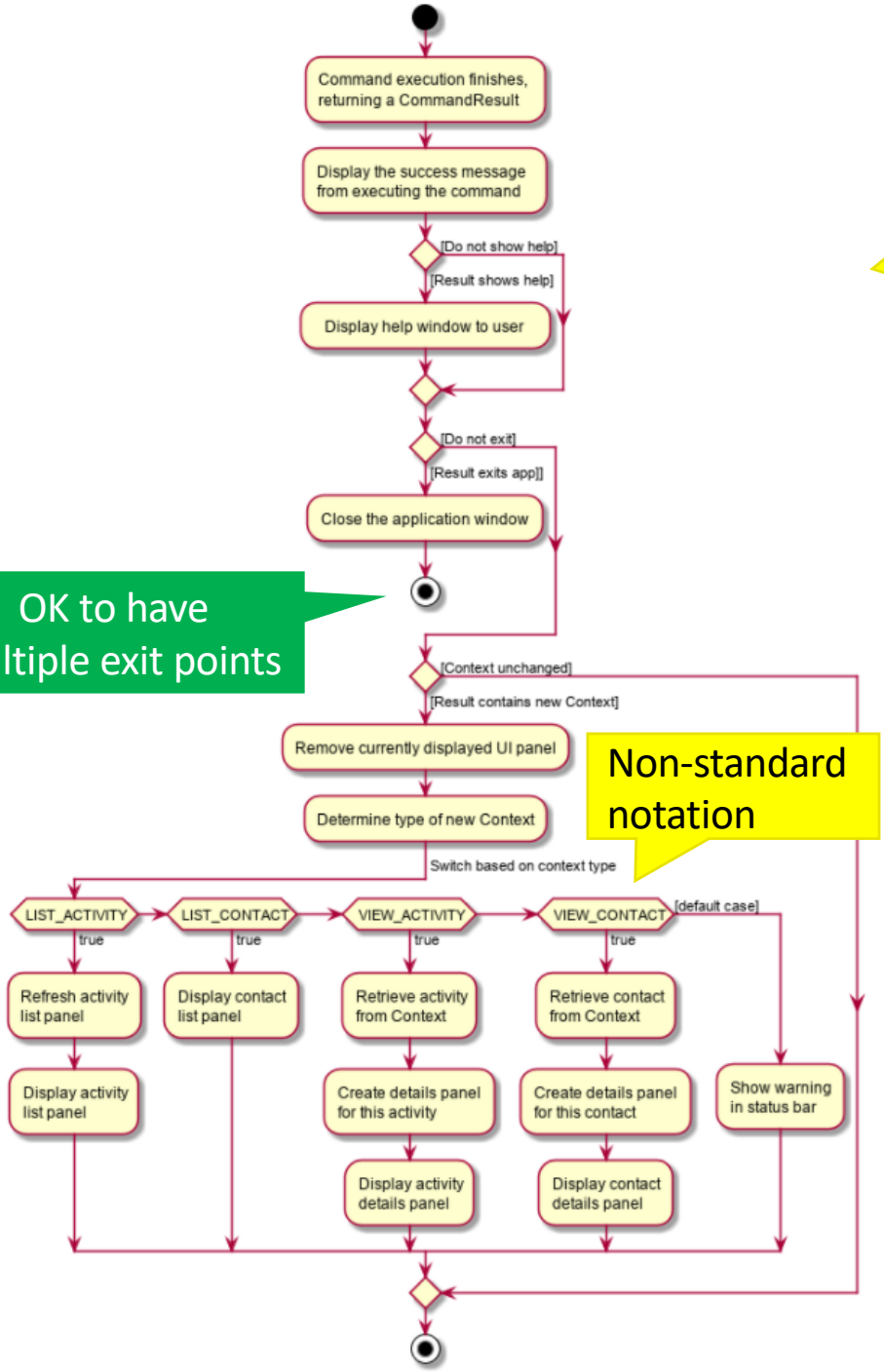


Should not continue lifeline after X but this is a limitation of PlantUML.

Figure 18. Sequence Diagram for Executing a `SplitCommand`

[Ex 10]

may be performed as a result of `MainWindow` parsing the `CommandResult`.



Too big. Break into two? (can use the rake symbol)

OK to have multiple exit points

Non-standard notation

Figure 16. Activity diagram for UI after successful command execution

[Ex 11]

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

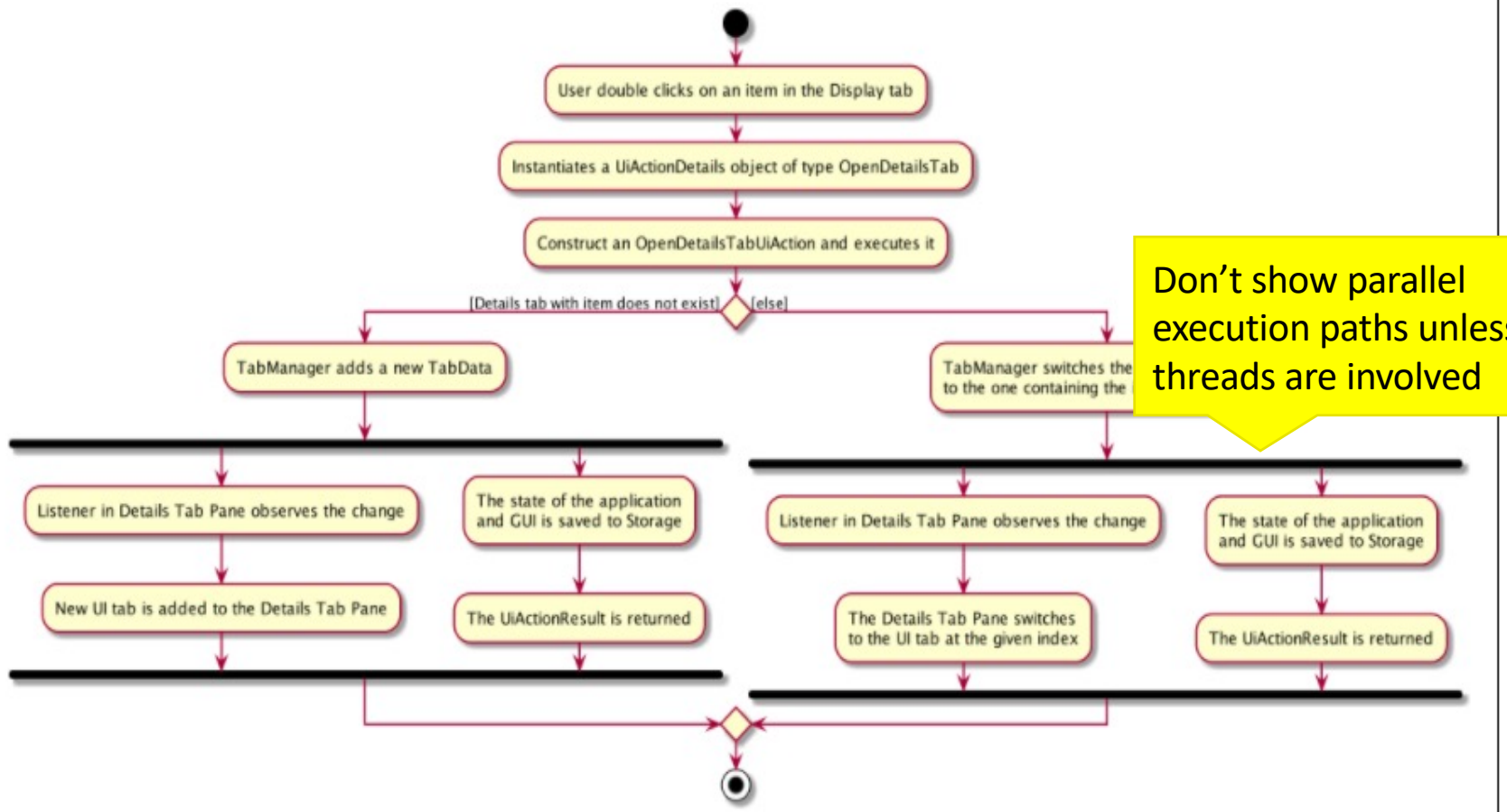


Figure 40. Activity Diagram for Opening a new Tab from the GUI

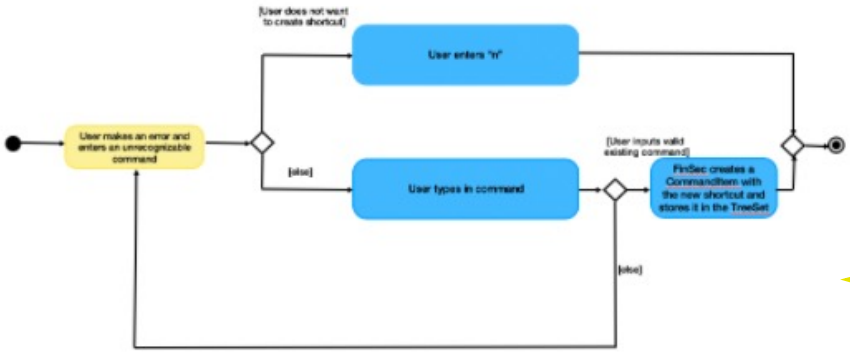
Don't show parallel execution paths unless threads are involved

[Ex 12]

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



Too small?

Looks like a screenshot. How to update this later?

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

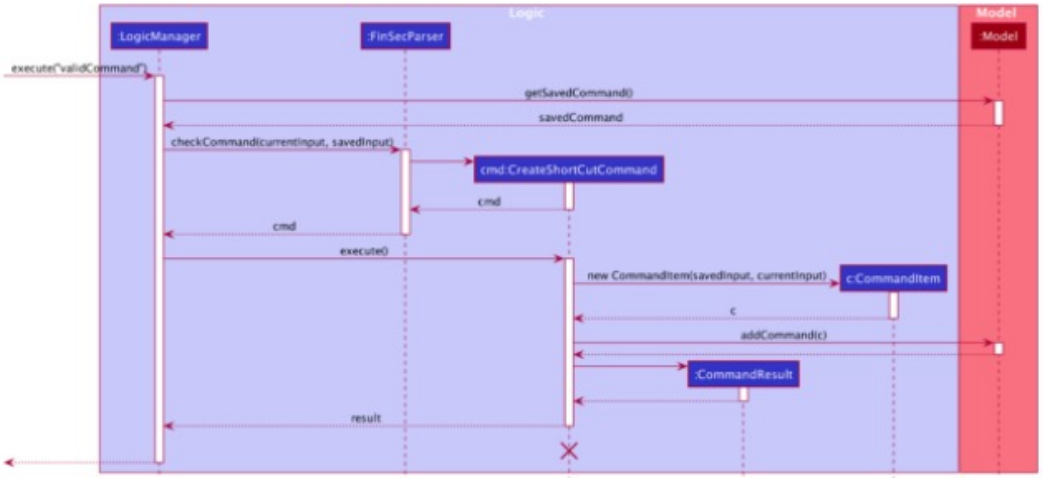


Figure 2.4.2: UML diagram when a user creates a shortcut

[Ex 13]

- `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 status claim. The order is as such: Pending, Approved, Rejected. There are 9 cases of comparison between 2 clai snippet below illustrates the comparator.

Too much code?

```
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;
```

[Ex 14]

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.

Should be 1a

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.

Duplication of similar info.

Better:

```
Similar to use case 7
except …
```