$AVE IT - Developer Guide

By: Team T10-3 Since: Mar 2020 Licence: MIT

1. Introduction

This section will introduce the intended use of the document and how to navigate it.

1.1. Purpose

The purpose of this document is to give insight to the implementation of the app, $AVE IT. This will give developers a better understanding on how they can work on the app in the future.

1.2. Usage

1.2.1. Navigate

Navigate the document by using `control - f` for Windows User or `command - f` for Mac Users. Alternatively, there is a clickable content page at the start of the document which you can click and bring you to the relevant sections.

2. Setting up

Refer to the guide here.

3. Design

3.1. Architecture

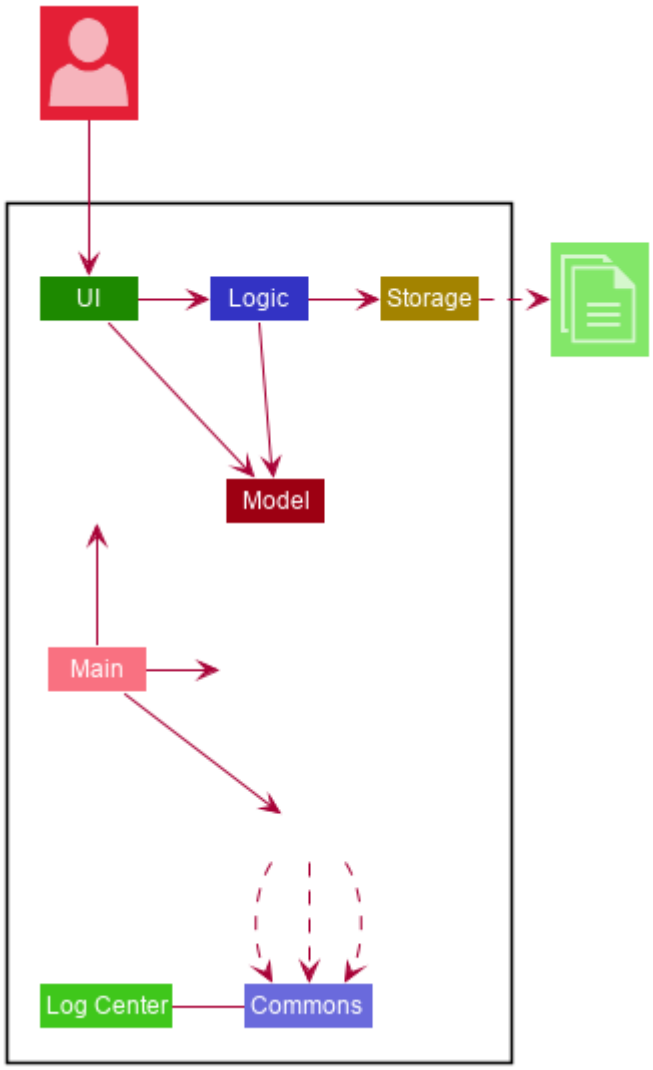

Figure 1. Architecture Diagram

The ***Architecture Diagram*** given above explains the high-level design of the App. Given below is a quick overview of each component.

> **TIP** The `.puml` files used to create diagrams in this document can be found in the diagrams folder. Refer to the Using PlantUML guide to learn how to create and edit diagrams.

`Main` has two classes called `Main` and `MainApp`. It is responsible for,

- At app launch: Initializes the components in the correct sequence, and connects them up with each other.
- At shut down: Shuts down the components and invokes cleanup method where necessary.

`Commons` represents a collection of classes used by multiple other components. The following class plays an important role at the architecture level:

- `LogsCenter` : Used by many classes to write log messages to the App's log file.

The rest of the App consists of four components.

- **`UI`**: The UI of the App.
- **`Logic`**: The command executor.
- **`Model`**: Holds the data of the App in-memory.
- **`Storage`**: Reads data from, and writes data to, the hard disk.

Each of the four components

- Defines its *API* in an `interface` with the same name as the Component.
- Exposes its functionality using a `{Component Name}Manager` class.

How the architecture components interact with each other

The *Sequence Diagram* below shows how the components interact with each other for the scenario where the user issues the command `exp delete 1`.

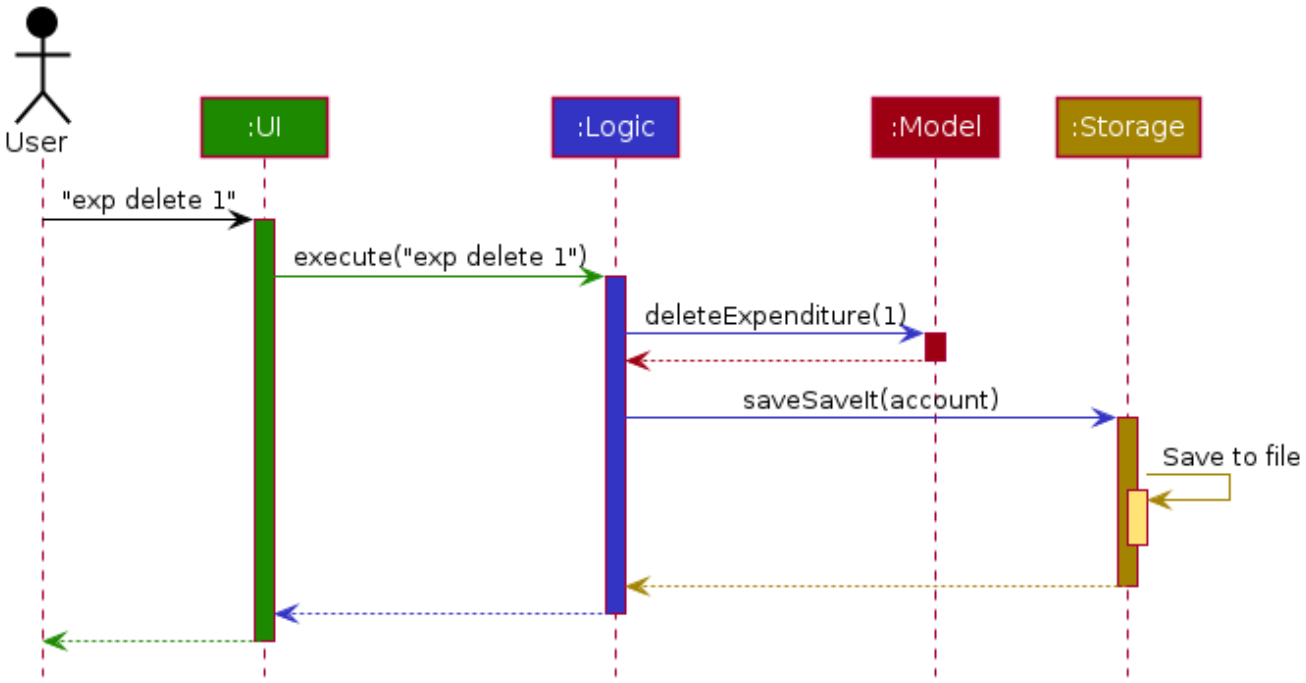

Figure 2. Component interactions for `exp delete 1` command

The sections below give more details of each component.

3.2. UI component

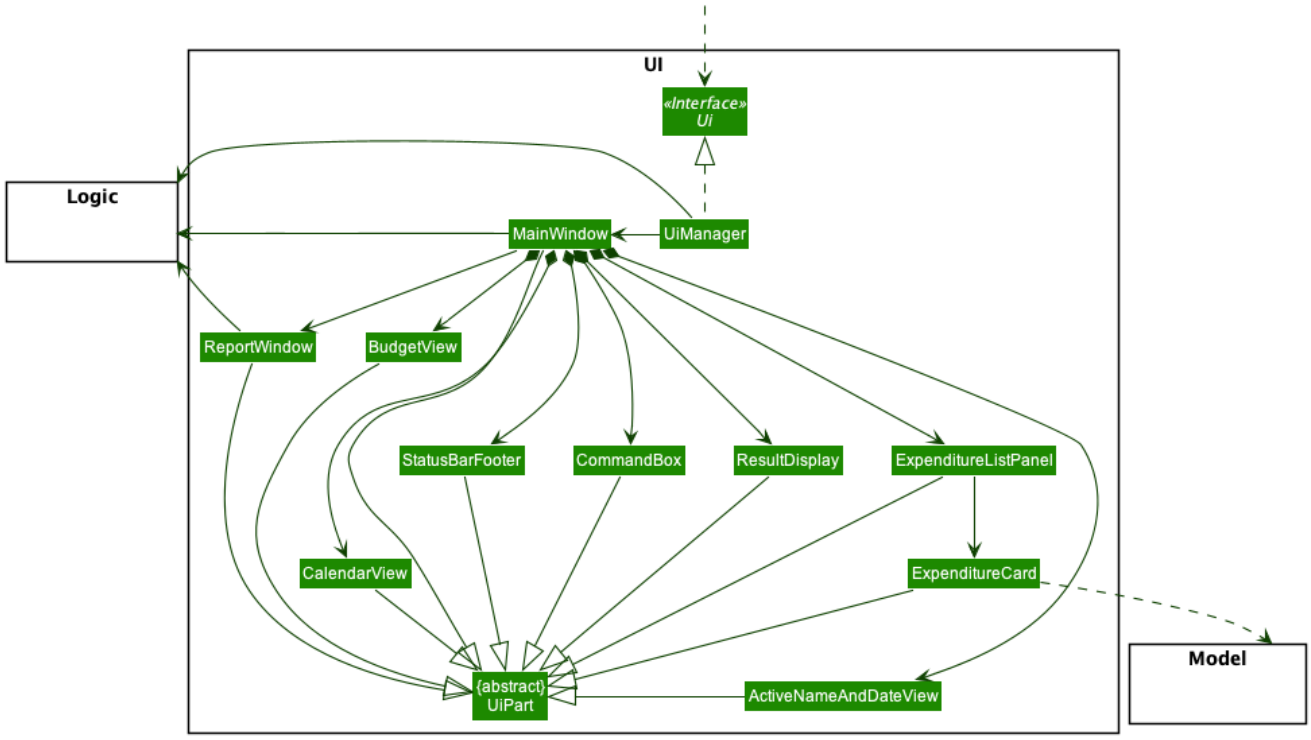

Figure 3. Structure of the Main UI Component

API : `Ui.java`

The Main UI consists of a `MainWindow` that is made up of parts e.g. `CommandBox`, `ResultDisplay`, `PersonListPanel`, `StatusBarFooter` , `Calendar` etc. All these, including the `MainWindow`, inherit from the abstract `UiPart` class.

The `UI` component uses JavaFx UI framework. The layout of these UI parts are defined in matching `.fxml` files that are in the `src/main/resources/view` folder. For example, the layout of the `MainWindow` is specified in `MainWindow.fxml`

The `UI` component,

- Executes user commands using the `Logic` component.
- Listens for changes to `Model` data so that the UI can be updated with the modified data.

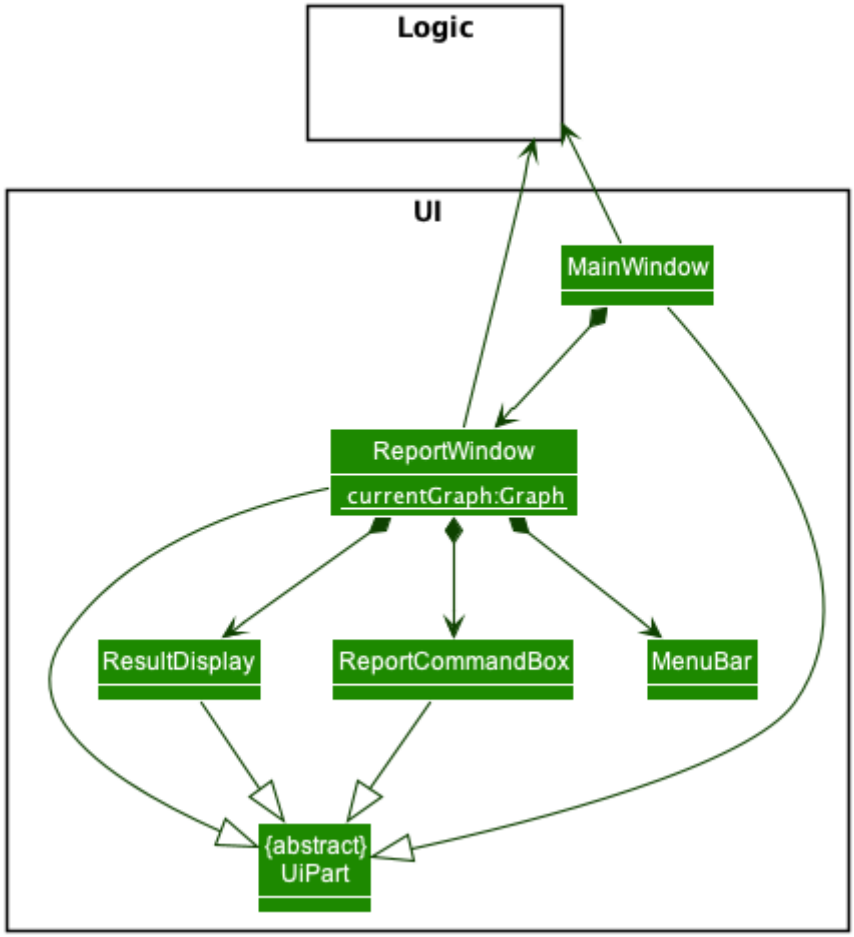

Figure 4. Structure of the Report UI Component

The Report UI consists of a `ReportWindow` that is made up of parts e.g. `ReportCommandBox` and `ResultDisplay` etc. The `ReportWindow` and `ReportCommandBox`, inherit from abstract `UiPart` class.

The `Report UI` component uses JavaFx UI framework. Layout of `ReportWindow` is defined in `.java` file that is in the `src/main/java/seedu/saveit/ui` folder. For example the layout of `ReportWindow` is specified in `ReportWindow.java`. Layout of `ReportCommandBox` is defined in the matching `.fxml` file that is in the `src/main/resources/view` folder. For example, the layout of the `ReportCommandBox` is specified in `ReportCommandBox.fxml`.

The `Report UI` component,

- Executes user commands using the `Logic` component.

3.3. Logic component

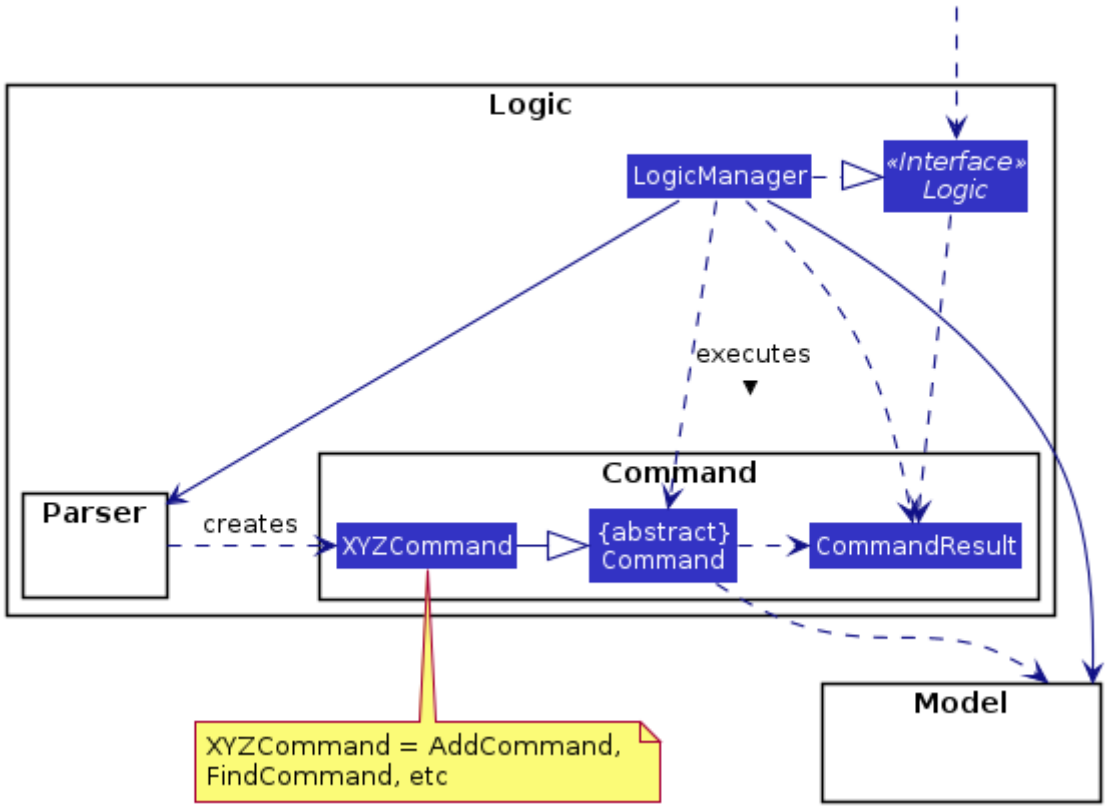

Figure 5. Structure of the Logic Component

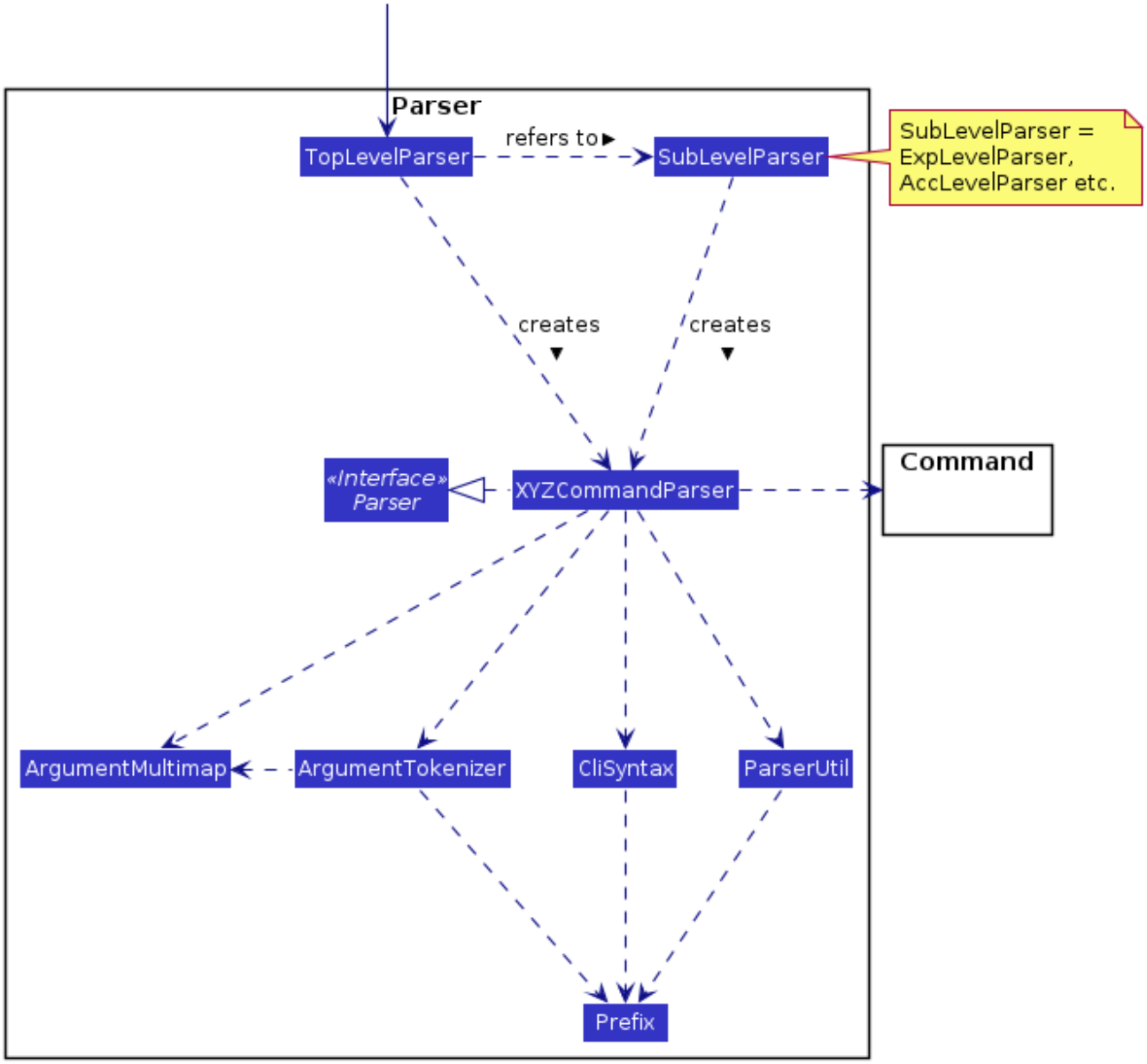

Figure 6. Structure of the Parser Component

API : `Logic.java`

1. `Logic` uses the `TopLevelParser` class to parse the user command from Main Window.
2. Depending on the command, the `TopLevelParser` class may use the `SubLevelParser` class e.g. `ExpLevelParser` to parse the command instead.
3. This results in a `Command` object which is executed by the `LogicManager`.
4. The command execution can affect the `Model` (e.g. adding a person).
5. The result of the command execution is encapsulated as a `CommandResult` object which is passed back to the `Ui`.
6. In addition, the `CommandResult` object can also instruct the `Ui` to perform certain actions, such as displaying help to the user.

Given below is the Sequence Diagram for interactions within the `Logic` component for the `execute("exp delete 1")` API call.

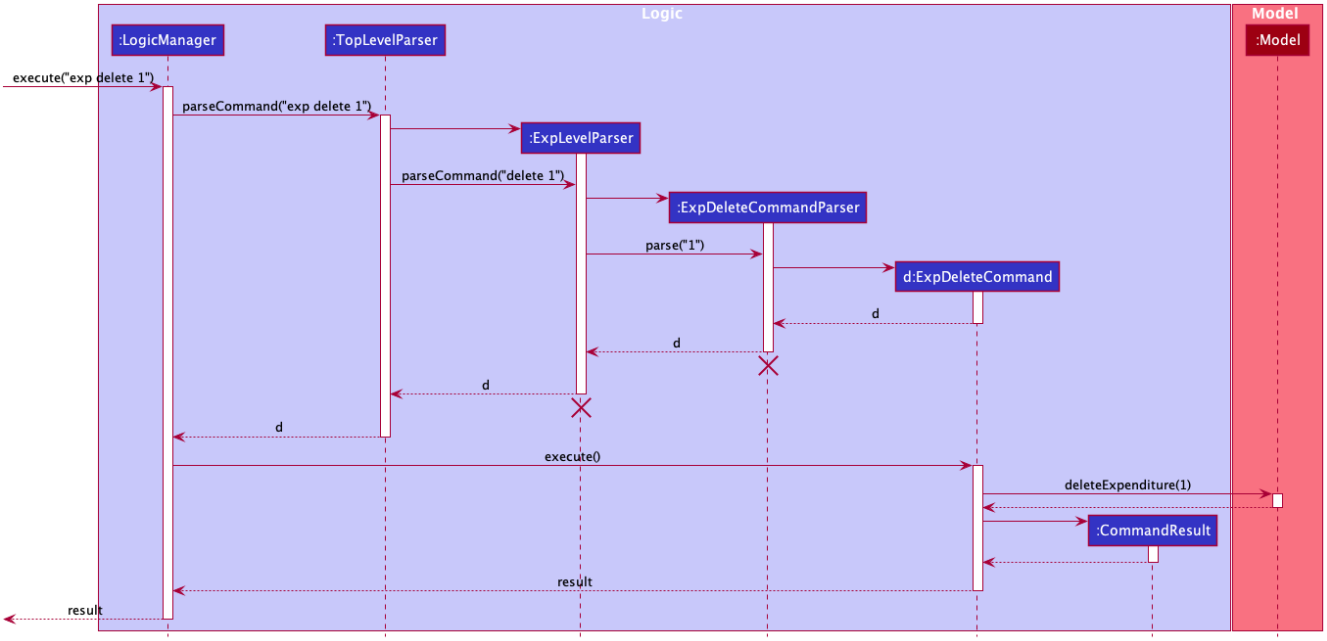

Figure 7. Interactions Inside the Logic Component for the `exp delete 1` Command

NOTE	The lifeline for `ExpDeleteCommandParser` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

3.4. Model component

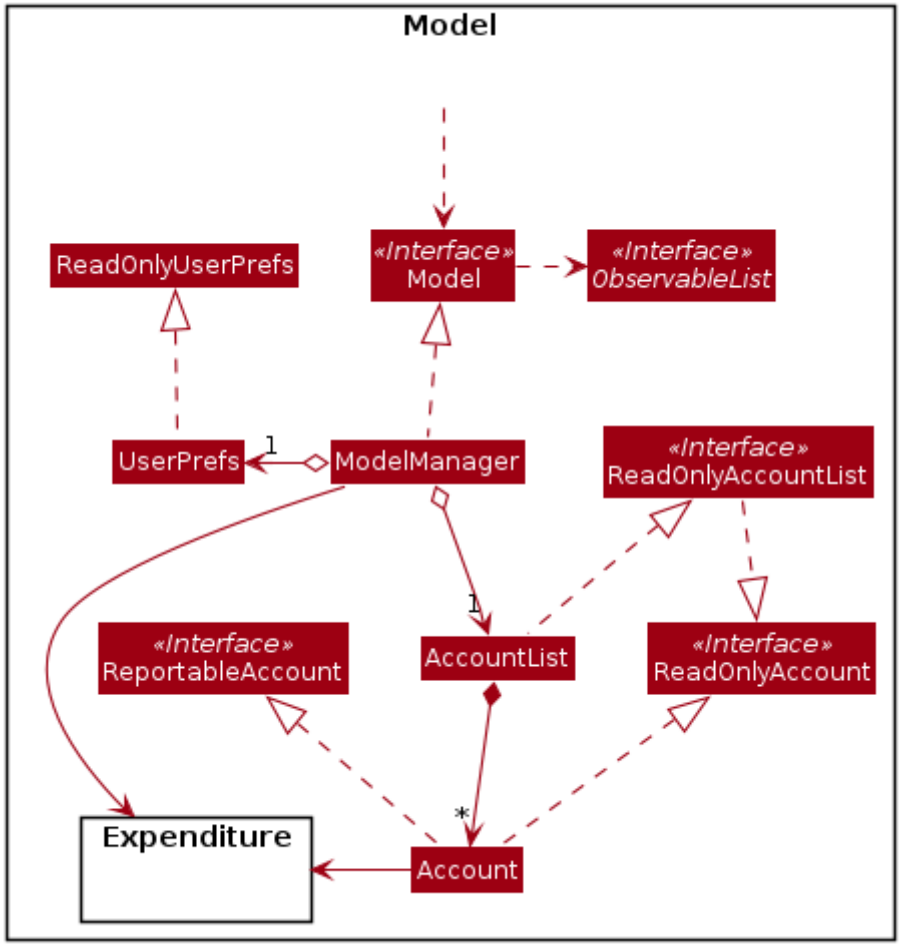

Figure 8. Structure of the Model Component

API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user's preferences.
- stores the `$AVE IT` data.
- exposes an unmodifiable `ObservableList<BaseExp>` that can be 'observed' e.g. the UI can be bound to this list so that the UI automatically updates when the data in the list change.
- does not depend on any of the other three components.

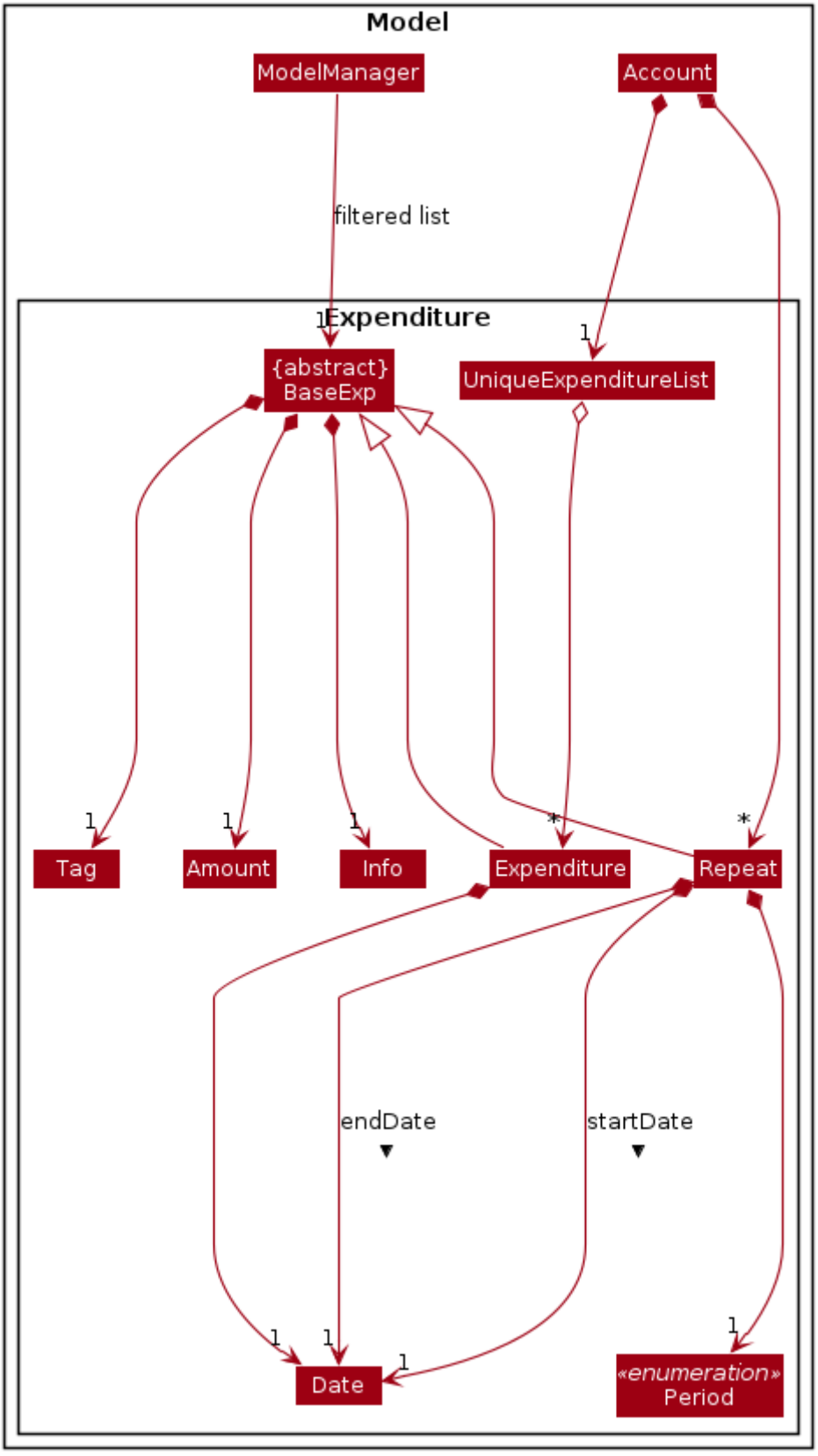

Figure 9. Structure of the Expenditure Component

The above image shows how the `expenditure` package interaction within itself, and other model components.

3.5. Storage component

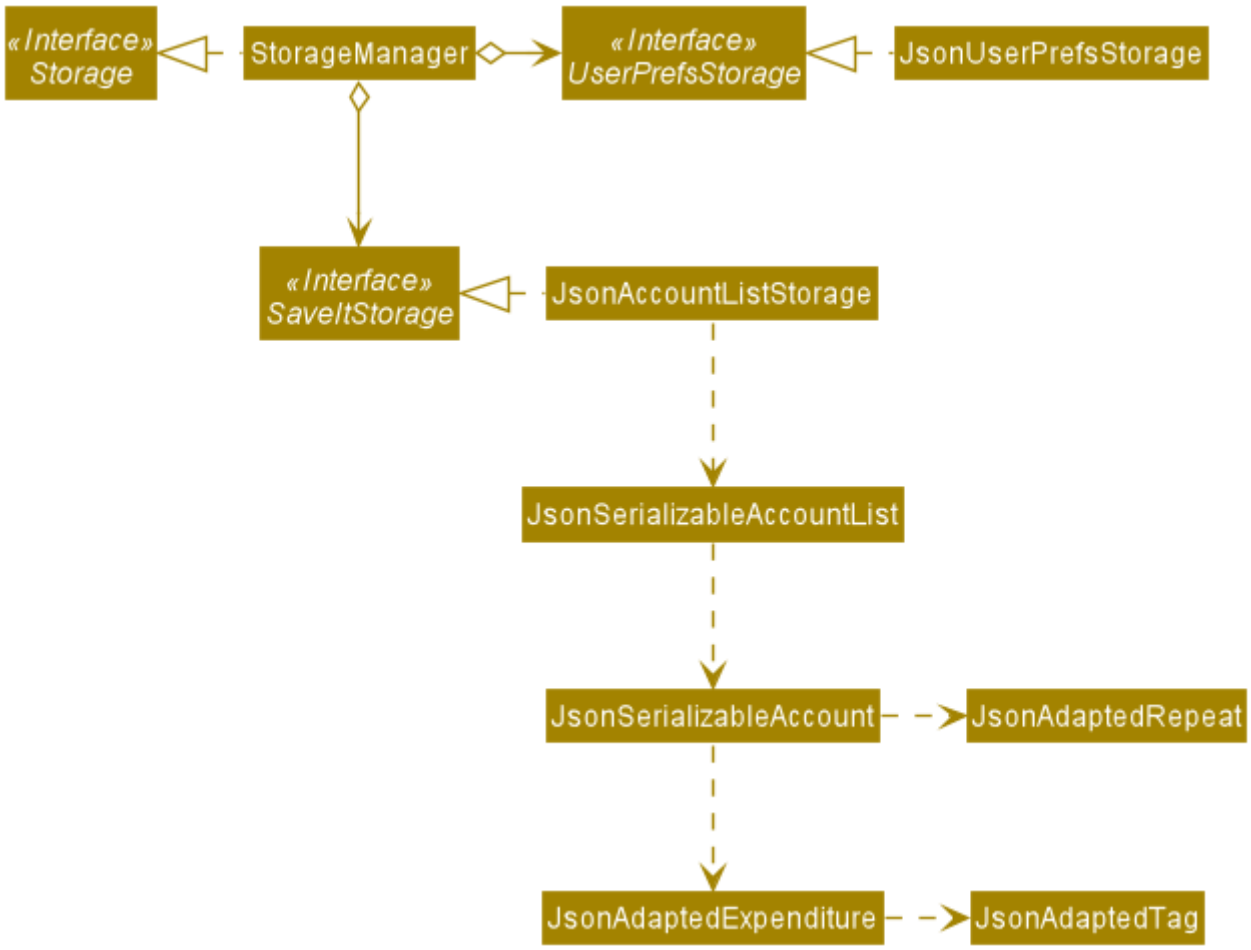

Figure 10. Structure of the Storage Component

API : `Storage.java`

The `Storage` component,

- can save `UserPref` objects in json format and read it back.
- can save the `$AVE IT` data in json format and read it back.

3.6. Common classes

Classes used by multiple components are in the `seedu.saveit.commons` package.

4. Implementation

4.1. Disjoint Account (Jiang Jiahui)

The disjoint accounts feature aims to help users better organise their expenditures by allowing them to separate the expenditures into different accounts.

4.1.1. Rationale

The user may be involved in different projects or have different roles which require expenditure tracking. Disjoint accounts aim to provide a higher degree of organization than just organising by date or tag.

4.1.2. Implementation

Below is a simplified class diagram that shows how the Account class relates to other classes (interfaces not shown).

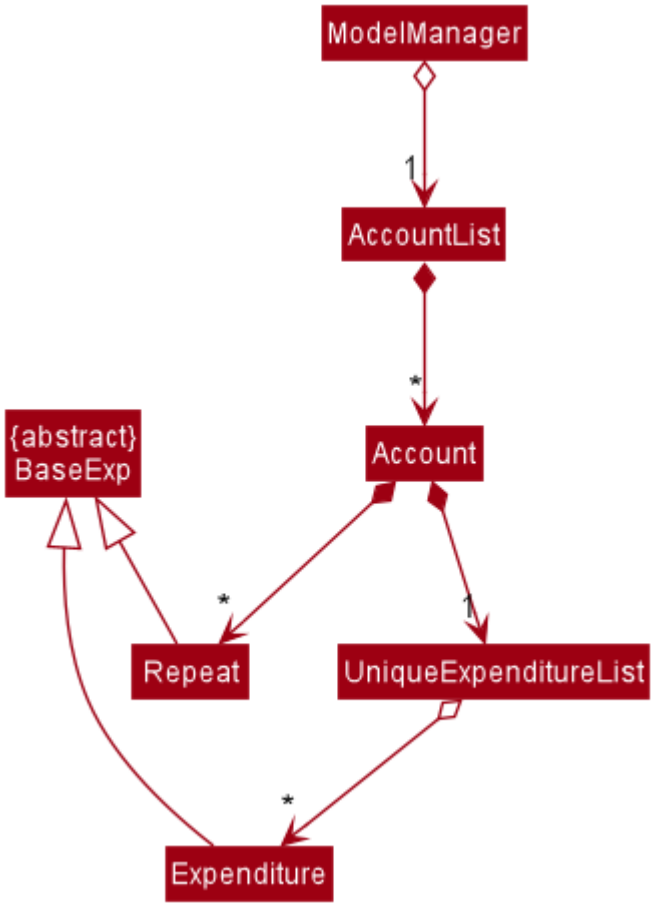

Figure 11. Simplified Account class diagram

Refer to Expenditure Implementation and Repeat Implementation for more details on these classes.

There are many commands that allow the user to add, delete, rename accounts and so on. Below is a sequence diagram that shows how a command to rename an account takes place.

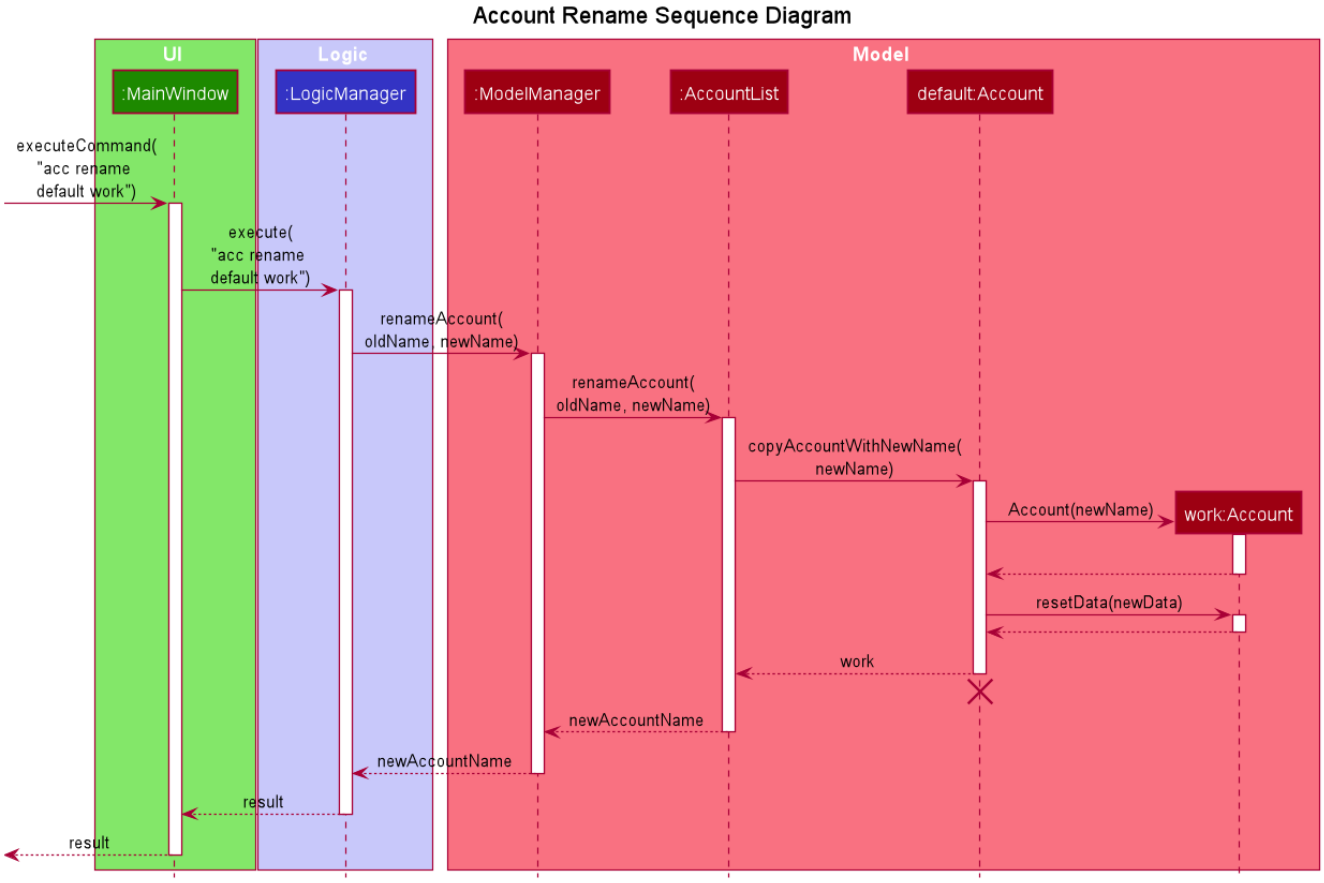

Figure 12. Sequence Diagram for execution of `acc rename` command

4.1.3. Design Consideration

This section contains some of our design considerations for the account feature.

Consideration: Storage of expenditures and repeats.

Alternatives	Pros	Cons
1. Store expenditures and repeats in the same list.	Fewer methods, since we can use the same method to add, edit or delete an expenditure/repeat.	Worse time complexity for some tasks which need to differentiate between expenditures and repeats.
2. **[current choice]** Store expenditures and repeats in separate lists.	Better time complexity for tasks such as calculation of total spending.	There needs to be double the number of getters, setters, methods to add, edit & delete the items.

Consideration: What to use for the backing list of the ListView UI component.

Alternatives	Pros	Cons

Exposing the repeat and expenditure lists in the accounts	Less troublesome when a repeat or expenditure command is executed.	More difficult to implement. Every time the active account is changed, the ListView has to be replaced as the backing list cannot be changed.
2. **[current choice]** Maintain a single list in the AccountList class as a backing list of the ListView.	There is no ambiguity as to which list is currently being displayed. This is safer as the lists in the accounts cannot be directly modified outside the Account class.	More troublesome when a repeat or expenditure command is executed, since both the active account and the list has to be updated.

4.2. Expenditure (Jiang Jiahui)

This is the most essential and basic feature of the application.

4.2.1. Rationale

The user can create expenditures to keep track of what they spend on, how much they have spent, and when it happens.

4.2.2. Implementation

Below is a class diagram that shows the Expenditure class and how it relates to other classes.

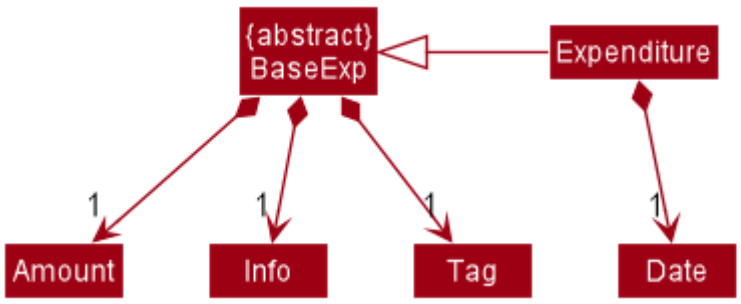

Figure 13. Expenditure class diagram

The activity diagram below shows what happens when the user enters an expenditure add command.

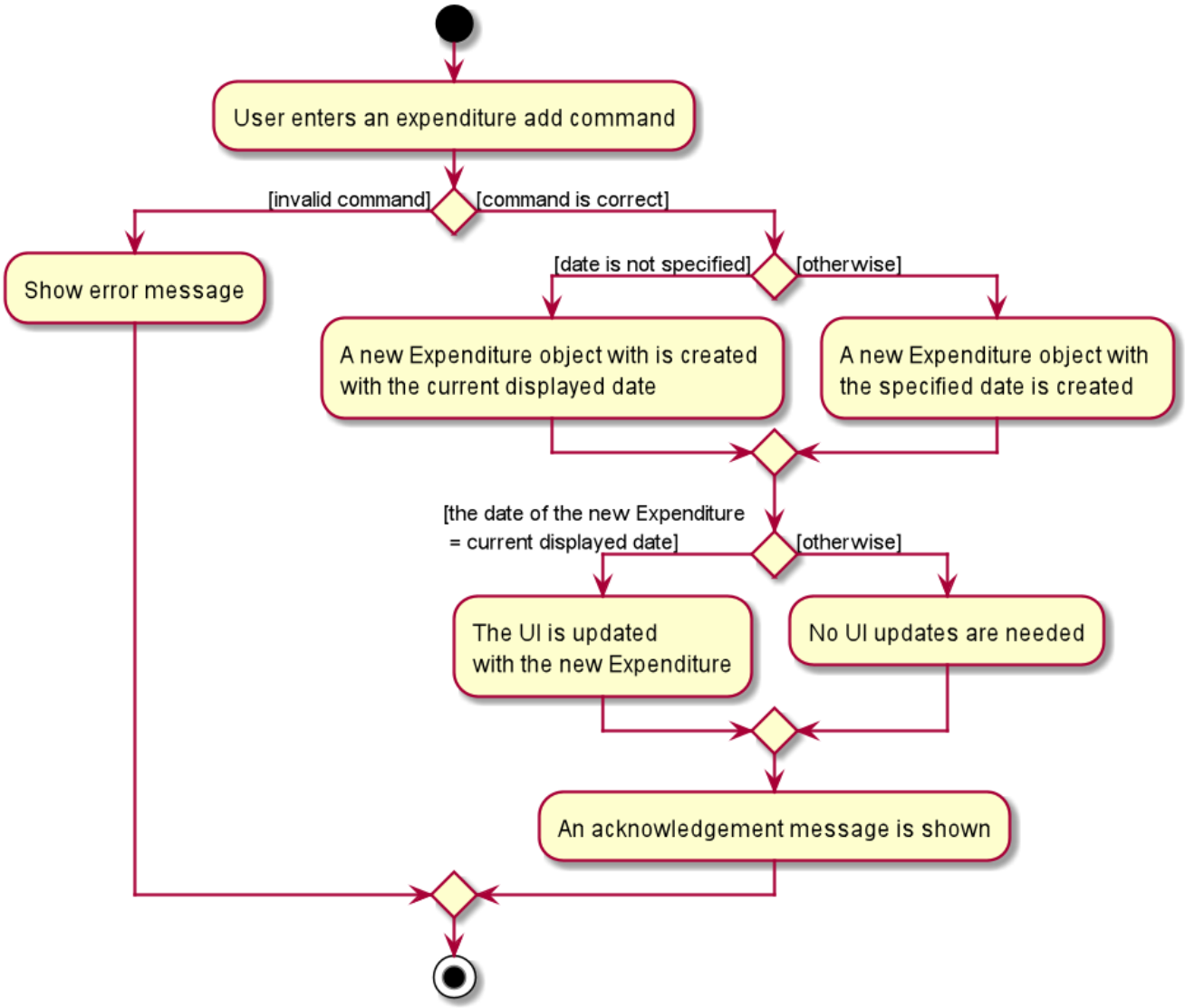

Figure 14. Expenditure add activity diagram

4.2.3. Design Consideration

Consideration: Number of tags an expenditure can have

Alternatives	Pros	Cons
One expenditure can have multiple tags.	More flexibility for the user.	More difficult to implement. This also makes it impossible to calculate total spendings for each tag due to possible overlapping.
2. **[current choice]** An expenditure has exactly one tag.	This makes it possible for the user to see total spending per tag using the Report feature.	Less flexibility for the user.

4.3. Recurring Expenditure feature `Repeat` (Zheng Shaopeng)

Recurring expenditure is one of the main features in `$AVE IT` and it is an expenditure automatically logged for user at their preferred frequency.

4.3.1. Rationale

`Repeat` allows user to keep track of expenditures that will occur either *daily*, *weekly*, *monthly* or *annually* without the need to key in the expenditures every day or month. Hence, this will provide more convenience for users as well as address the need for such a feature since recurring expenditures are common. For example, day to day commuting expenses.

4.3.2. Implementation

Below is a class diagram shows different components that `Repeat` contains.

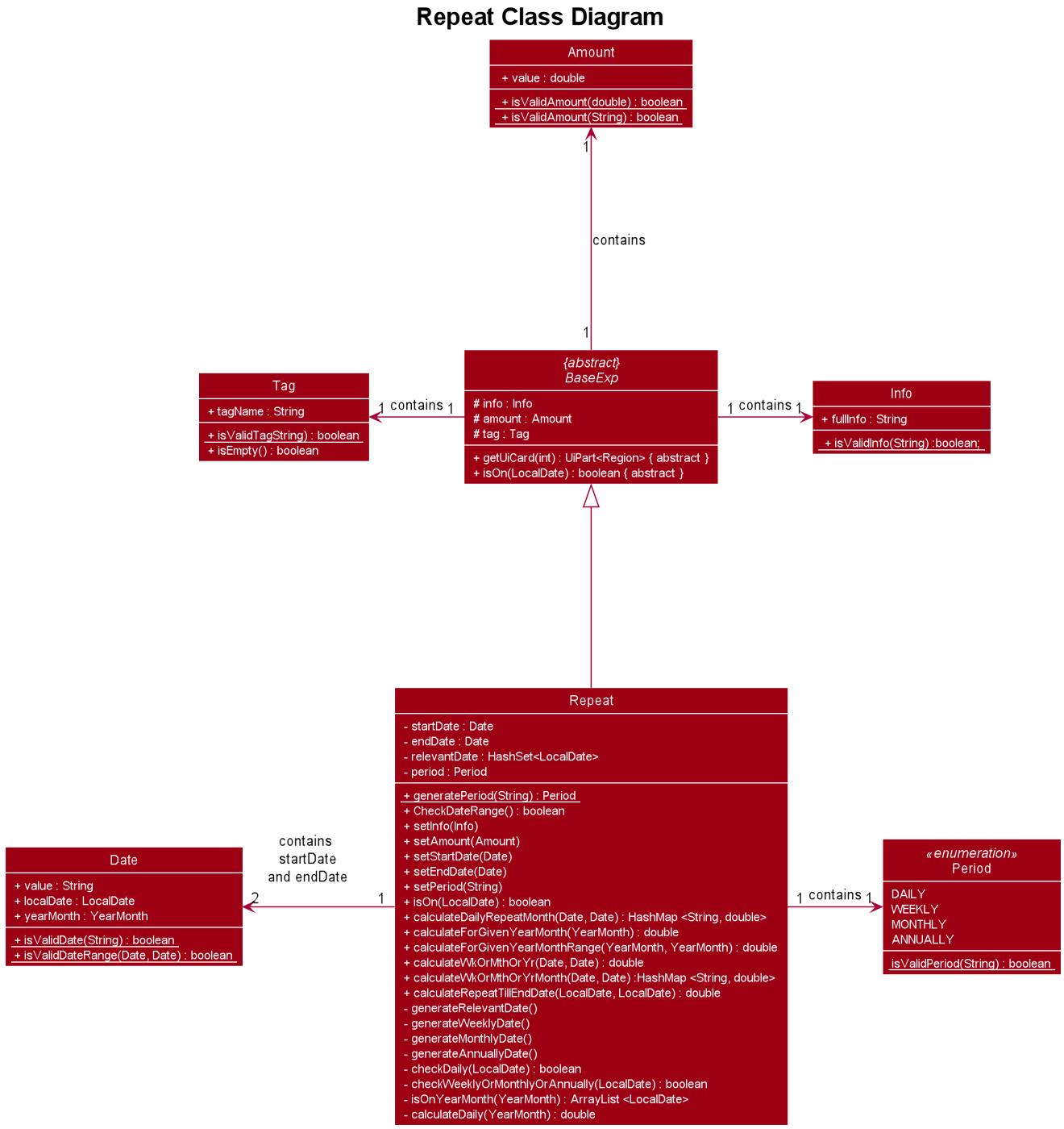

Figure 15. Class diagram for showing what `Repeat` consist.

For each account, it has its own `list` which all the `Repeat` objects are stored. There are different types of command that is cater for `Repeat` such as add, edit and delete. The following activity diagram shows what how a `Repeat` can be added.

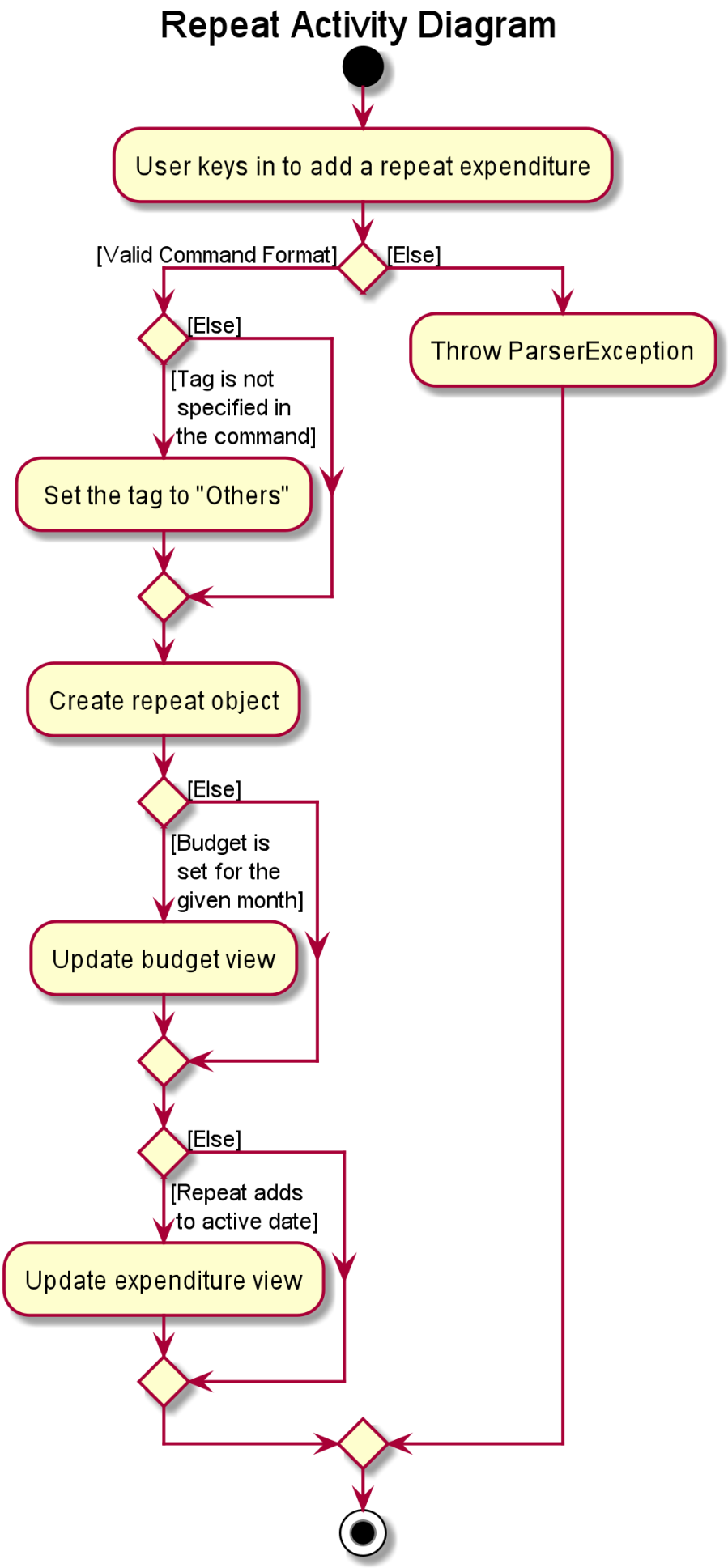
Repeat Activity Diagram
User keys in to add a repeat expenditure
[Valid Command Format]
[Else]
Throw ParserException
[Else]
[Tag is not specified in the command]
Set the tag to "Others"
Create repeat object
[Else]
[Budget is set for the given month]
Update budget view
[Else]
[Repeat adds to active date]
Update expenditure view

Figure 16. Activity diagram for `repeat add`

4.3.3. Design Consideration

Alternatives	Pros	Cons
(Current choice) Have a repeat class which extends `BaseExp`.	Able to mass delete and edit all the expenditures under this `Repeat` easily.	Hard to implement, especially when we have to calculate monthly spending so to generate report and statistics.
Mass operation: add `Expenditure` object to all those dates which state in the command.	Easy to implement.	User are unable to edit all the expenditures which are recurring. Users have to delete such expenditures one by one.

4.4. Budget (Lim Feng Yue)

Budget feature allows user to input their budget for any month, and calculates the balance from the total spending. Depending on the amount of balance and whether the budget is set, different piggy bank images will be shown.

4.4.1. Rationale

As the application is about budget management and expenditure tracking. Budgeting is an essential feature to allow user to keep track on how they are spending their money.

4.4.2. Implementation

The budget feature consists of using a command, and a part of the UI display.

The following activity diagram shows what happens to the `BudgetView` which displays the budget details when a command is entered.

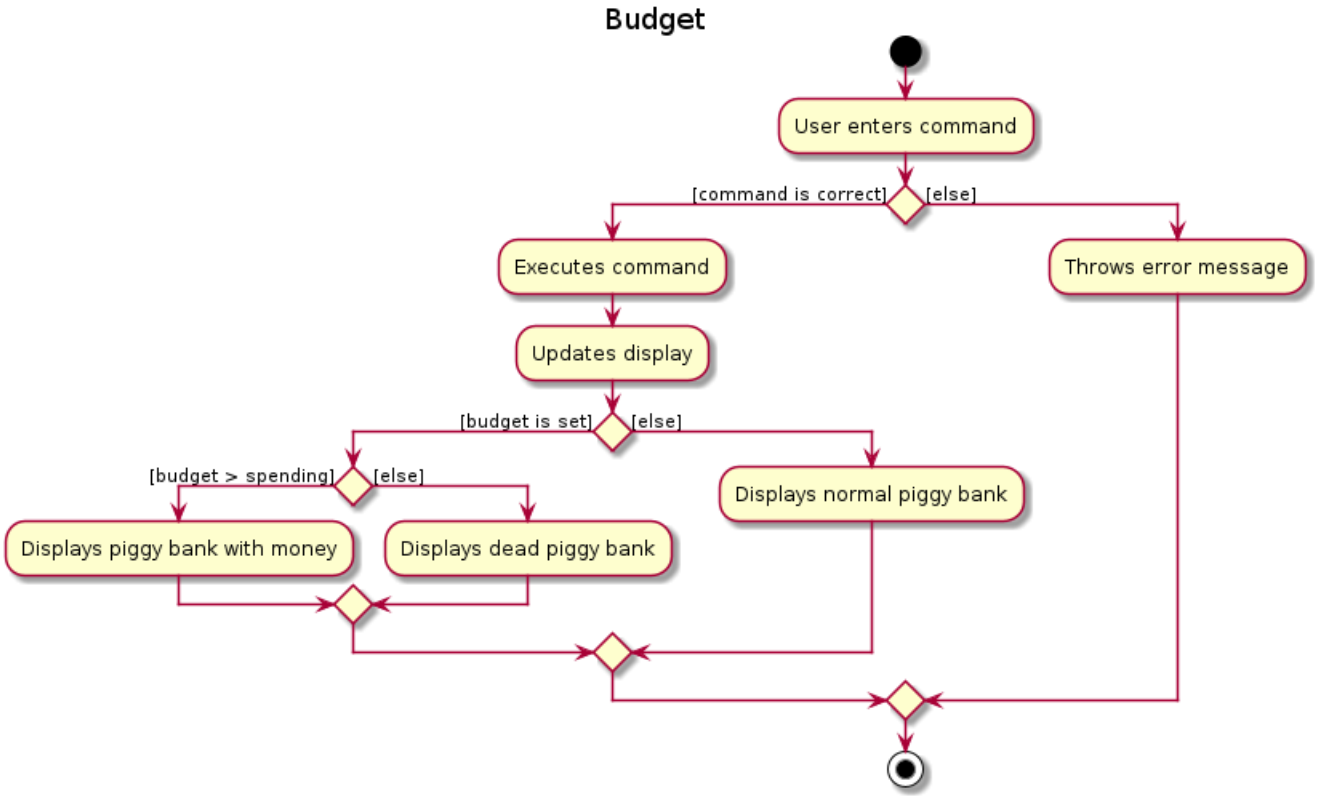

Figure 17. Activity Diagram of Budget View

The implementation of setting the budget of the month is through the command format of `setbudget -a AMOUNT -ym YEAR_MONTH`. The process of how the command is parsed is shown below using an example, `setbudget -a 123 -ym 2020-04`.

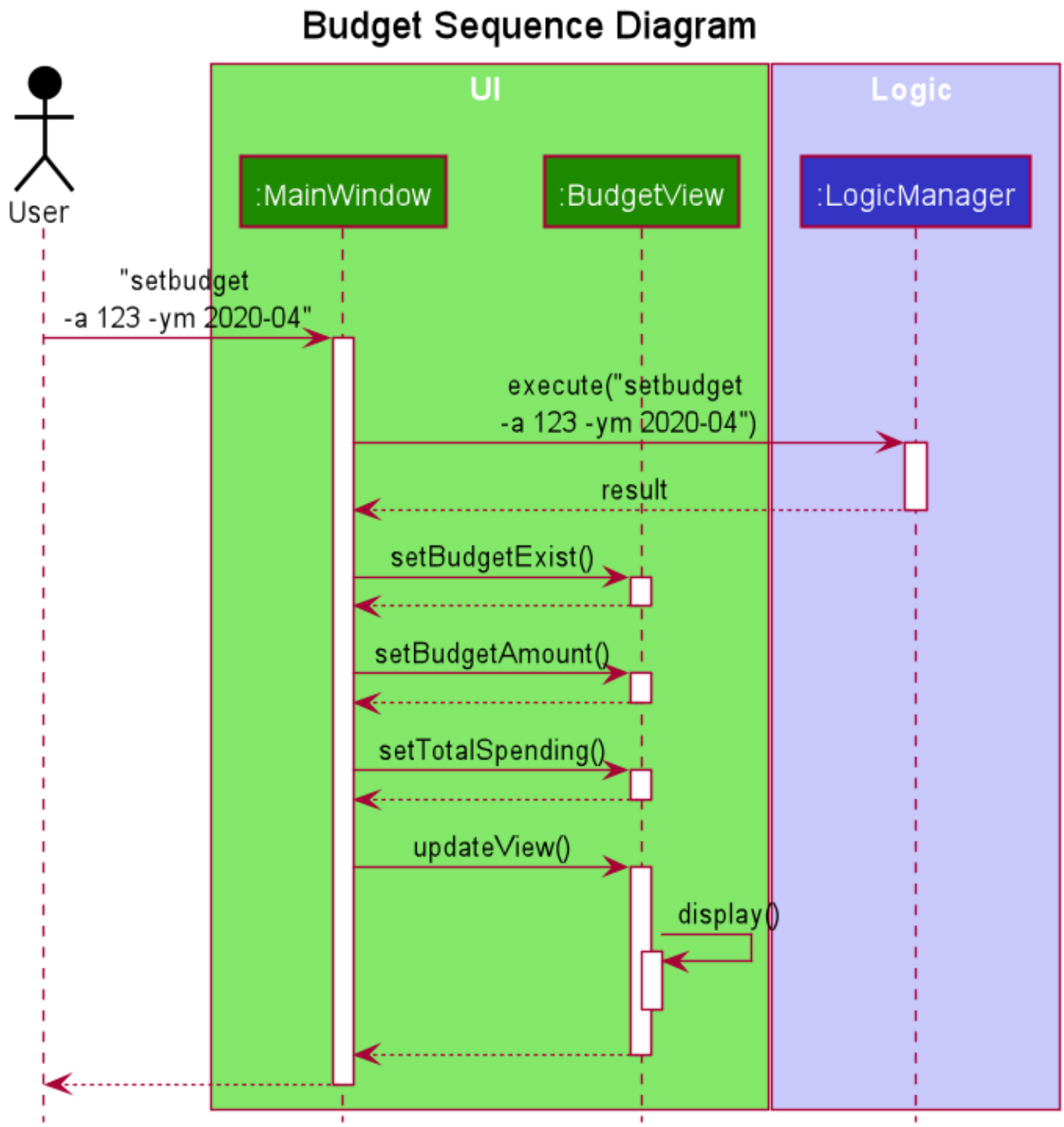

Figure 18. Set Budget Sequence Diagram (UI and Logic)

The above sequence diagram shows the interaction of the user and the UI. After entering the command, the `BudgetView` will be updated using the result returned by the `LogicManager`.

The information displayed are:

1. The budget amount, e.g. `$123.00`
2. The total spending in `2020-04` (month)
3. The balance, which is the difference between the budget amount and the total spending
4. An image as visual feedback

The sequence diagram below shows a more detailed view of what happens inside the `LogicManager`.

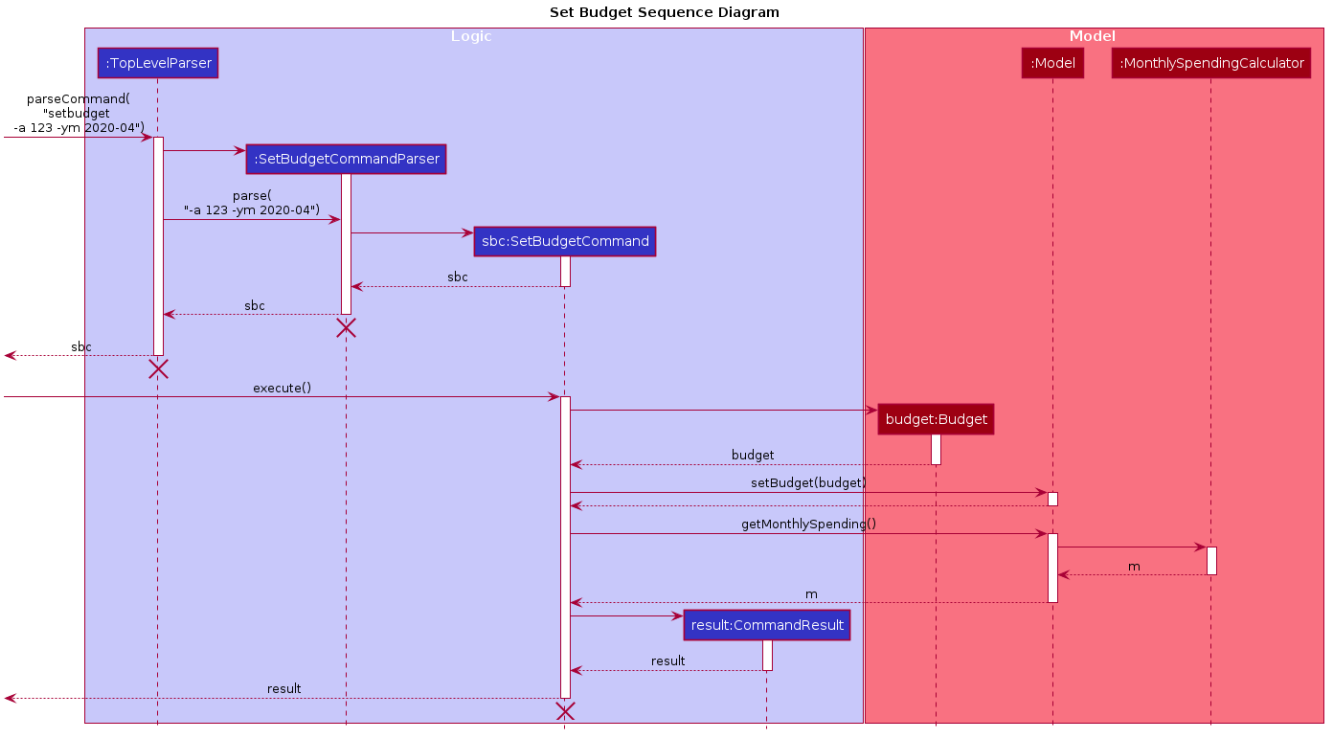

Figure 19. Set Budget Sequence Diagram (Logic and Model)

4.4.3. Design Consideration

Aspect: Calculation of Budget

Alternatives	Pros	Cons
[current choice] Budgets are set monthly only.	- Most common budget setting type. - Easy to implement.	- It is not useful for users who prefer other kinds of calculation of budget.
Variability in how budget is calculated, e.g. weekly, monthly, yearly.	- Gives users more choice on how they want to budget.	- Way more difficult to implement.

Aspect: Visual Display of Budget

Alternatives	Pros	Cons
[current choice] Display 3 states of budget balance in image.	- Easier to see if the budget is being met.	- Requires a bit more code, and finding images.
No visual display, just text display.	- Very easy to implement.	- The UI may look a bit plain.
Better UI display, showing different variations of whether budget is met e.g. a chart.	- Gives users better insight on how they are handling their budget.	- More work is required.

4.5. Report (Ng Xinpei)

Report is one of the main features in `$AVE IT` and its purpose is to collate and show users their expenditure breakdowns.

4.5.1. Rationale

The report feature is an important feature that allows users to see their expenditure breakdowns within a certain time period. Currently, the expenditure are categorise in terms of tags and this will give users a clear overview of what they are spending on over this period of time.

4.5.2. Implementation

The report feature can be accessed from 2 platforms either **Main Window** or **Report Window**

- **Main Window**

The input from user is parsed using a specific command parser depending on which of the 3 commands: `report view`, `report print` and `report export` were inputted by user.

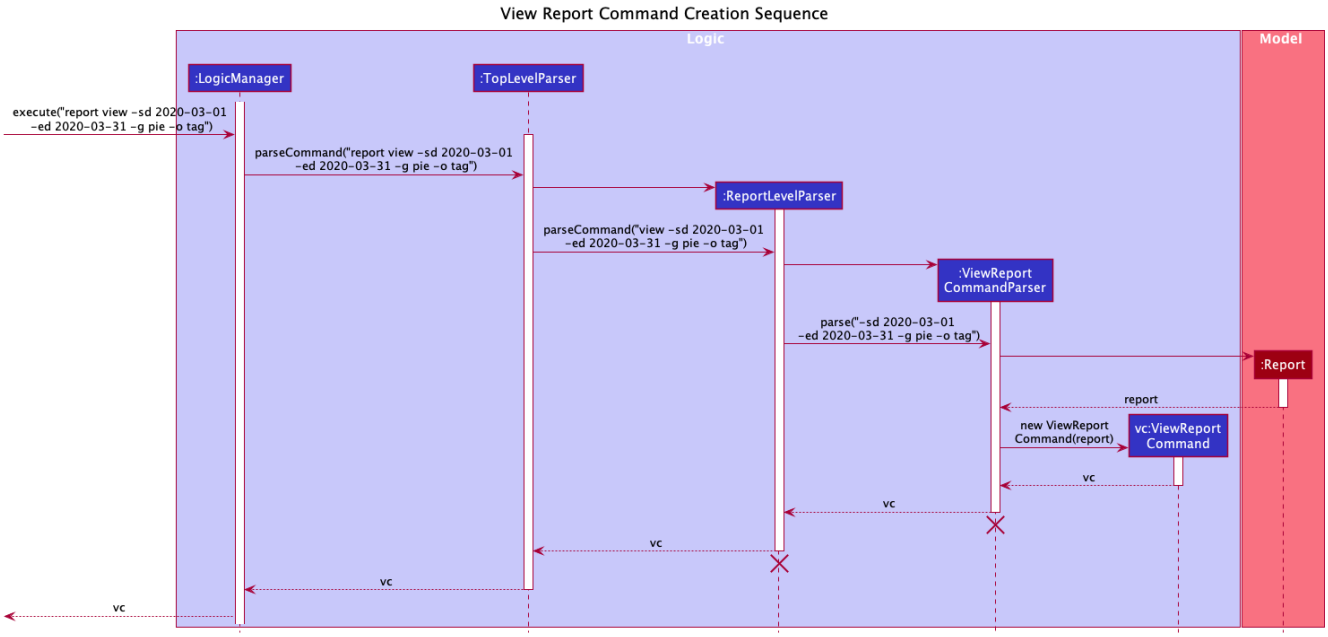

Figure 20. Sequence Diagram for View Report Command Creation

If a valid `report view` command was input, the *ViewReportCommandParser* will parse the input and convert the Strings : start date, end date, graph type and organisation into *Date* , *Report.GraphType* and *organisation(String)* object respectively. These objects are used to create the *ViewReportCommand* object. The figure above shows how the different objects interact to create *ViewReportCommand* object.

As shown in the figure above, when a user inputs a valid `report view` command:

1. `report view` command will be parsed and a new *Report* object will be created.
2. A new *ViewReportCommand* object containing the *Report* object will be created.

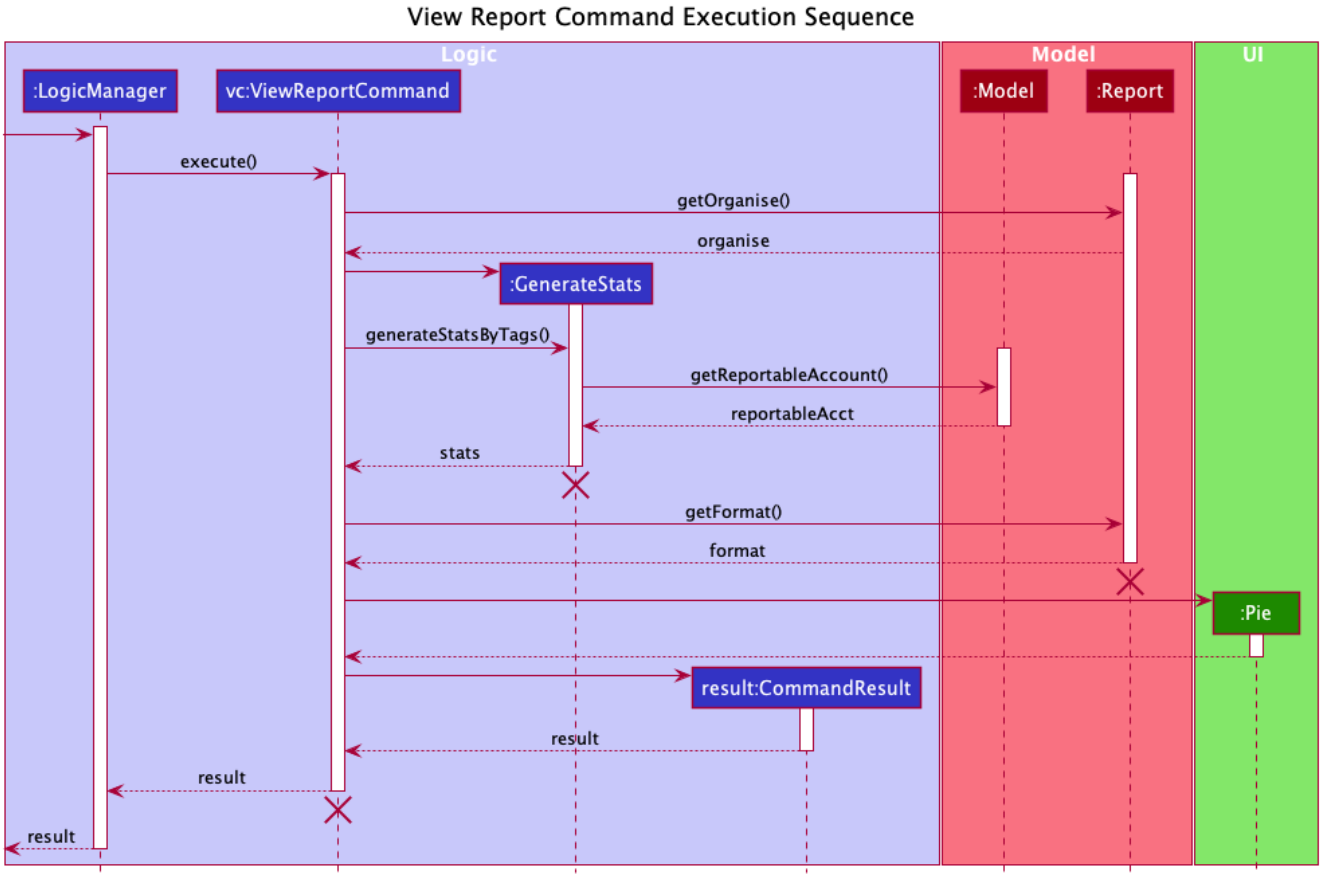

Figure 21. Sequence Diagram for View Report Command Execution

The *ViewReportCommand* object will be executed. The result of the execution is popping out of *ReportWindow* which will showcase a expenditure breakdown report. The figure above shows how the objects interact to execute *ViewReportCommand* object.

As shown in the figure above,

1. The *ViewReportCommand* object will be executed and a new *GenerateStats* object will be created.
2. The *GenerateStats* object will calculate and generate statistics from *ReportableAccount* object requested from *Model* through `getReportableAccount` method.
3. A new *Pie* object will be created.
4. A new *CommandResult* will be constructed and returned.
5. Report Window will pop out.

If a valid `report print` command was input, the *PrintReportCommandParser* will parse the input and convert the Strings : start date, end date, graph type into *Date*, *Report.GraphType* and *organisation(String)* object respectively. These objects are used to create *PrintReportCommand* object which will be executed. The result of execution will be sending a print job to your printer, printing out the report.

If a valid `report export` command was input, the *ExportReportCommandParser* will parse the input and convert the Strings : start date, end date, graph type into *Date*, *Report.GraphType* , *organisation(String)* and *file name(String)* object respectively. These objects are used to create *ExportReportCommand* object which will be executed. The result of execution will be exporting a

PNG version of the report into your computer with file name.

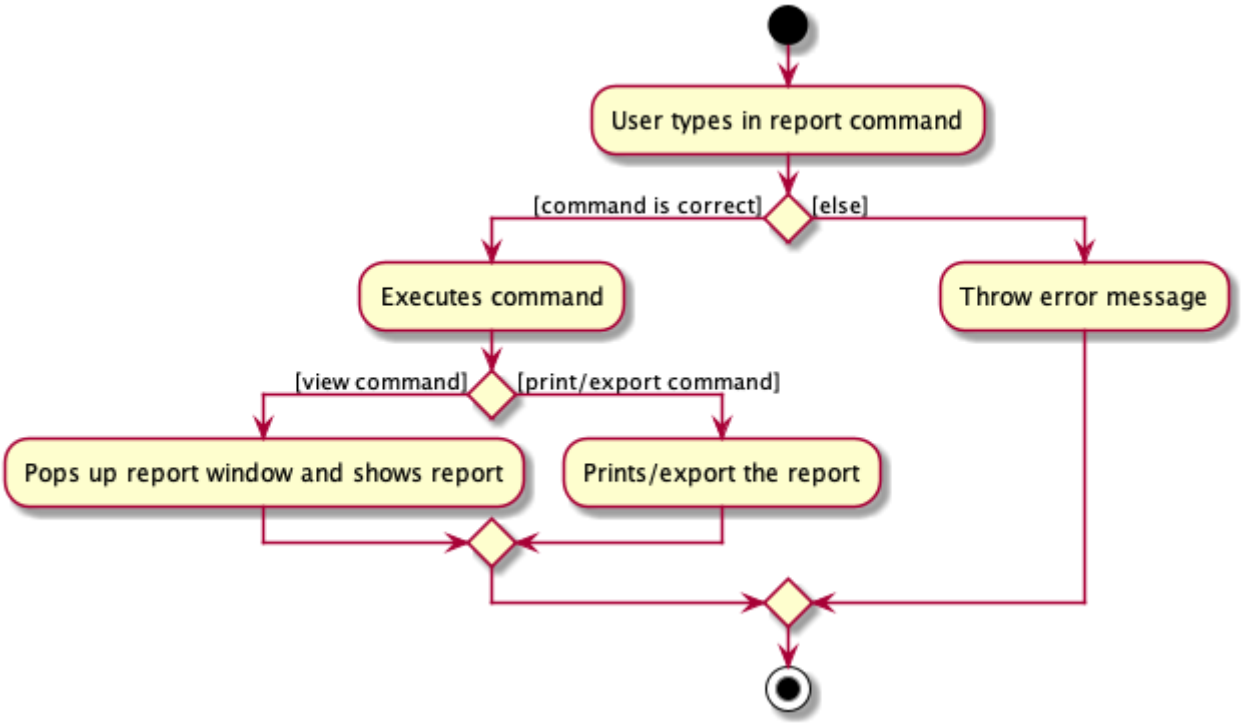

Figure 22. Activity Diagram for Report in Main Window

The activity diagram summarises what can happen when user enter a `report` command in the **Main Window**

- **Report Window**

The **Report Window** can be accessed via the *Report button* in the **Main Window** or via `report view` command.

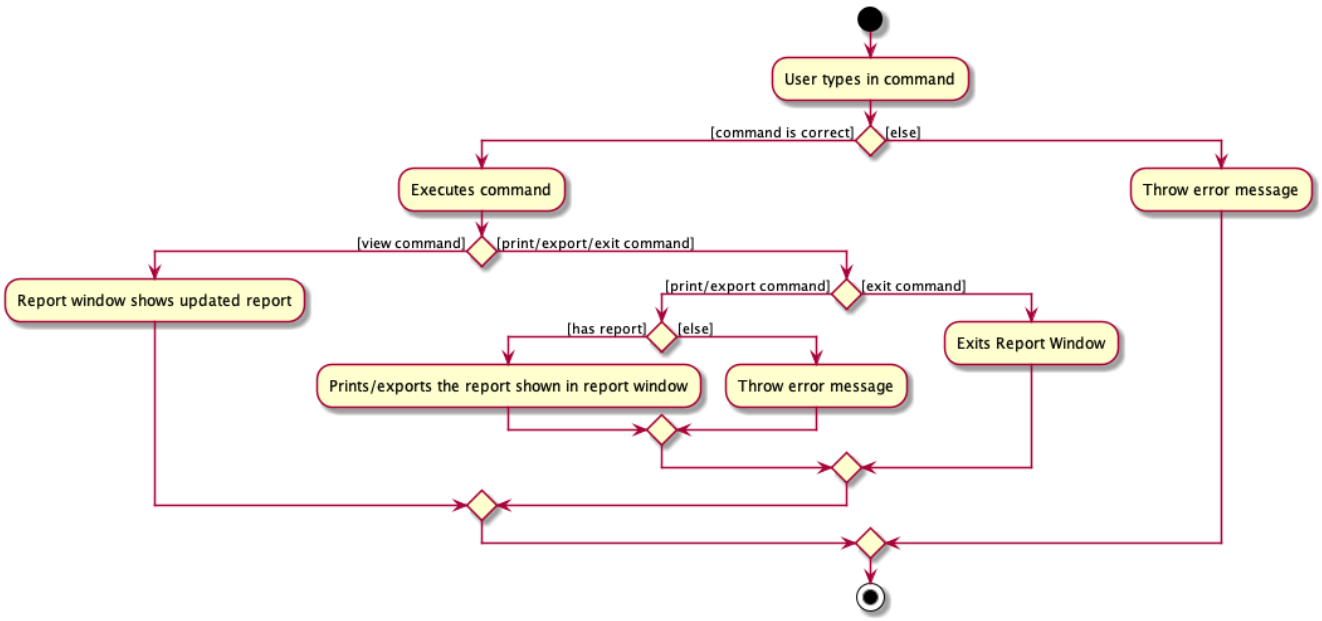

Figure 23. Activity Diagram for Report Window

The activity diagram shows what can happen when users enter in a new command in the **Report Window**.

4.5.3. Design Considerations

Consideration : Minimal changes to current UI implementation, especially Main Window

- Pros: Less dependency with current working code for Main Window. Hence, even if report feature fails, it is likely that Main Window can continue running.
- Cons: It could be more complicated to implement.

Consideration : Avoid cluttering the Main Window UI

- Pros: Better user experience.
- Pros: As report is not using any space in Main Window, we could introduce and showcase other smaller and useful features in the Main Window UI.
- Cons: It could be more complicated to implement

Due to the above considerations, we implemented report viewing in a pop up window.

4.5.4. NOTE

1. While it is allowed for users to state any date range for report and have any number of tags, it is recommended to them that to keep the **date range within 12 months** for **reports generated by months** or **keep number of tags to be within 12** for **reports generated by tags** due to sizing issues.
2. Overlaps can occur in PieChart when there are huge differences in expenditure values, hence it is recommended to users that Bar Chart should be used in such instances.
3. While it is allowed for users to send multiple print jobs, users are recommended to avoid sending multiple print jobs in a short interval. The window may be unresponsive if the default printer is not set up, this will last for a few seconds. Eventually, a response stating "Set available printer as default printer before printing" will be given.
4. Only months or tags with total spending of more than 0 dollars will be reported.
5. The Report folder is located at the directory where you run the jar file.

4.6. Calendar feature (Zheng Shaopeng)

4.6.1. Rationale

Calendar is a feature that has a clickable calendar which users can use to navigate between the different days. It also shows the date of the expenditures the user is viewing as well as today.

4.6.2. Implementation

The implementation of the above functions will be described separately in this section.

The users are given two different choice on how to navigate between the days:
1. UI interaction with the calendar view.
2. Make use of `go YYYY-MM-DD` command.

The following sequence diagram shows you how the `go YYYY-MM-DD` (E.g. `go 2020-04-01`) command works.

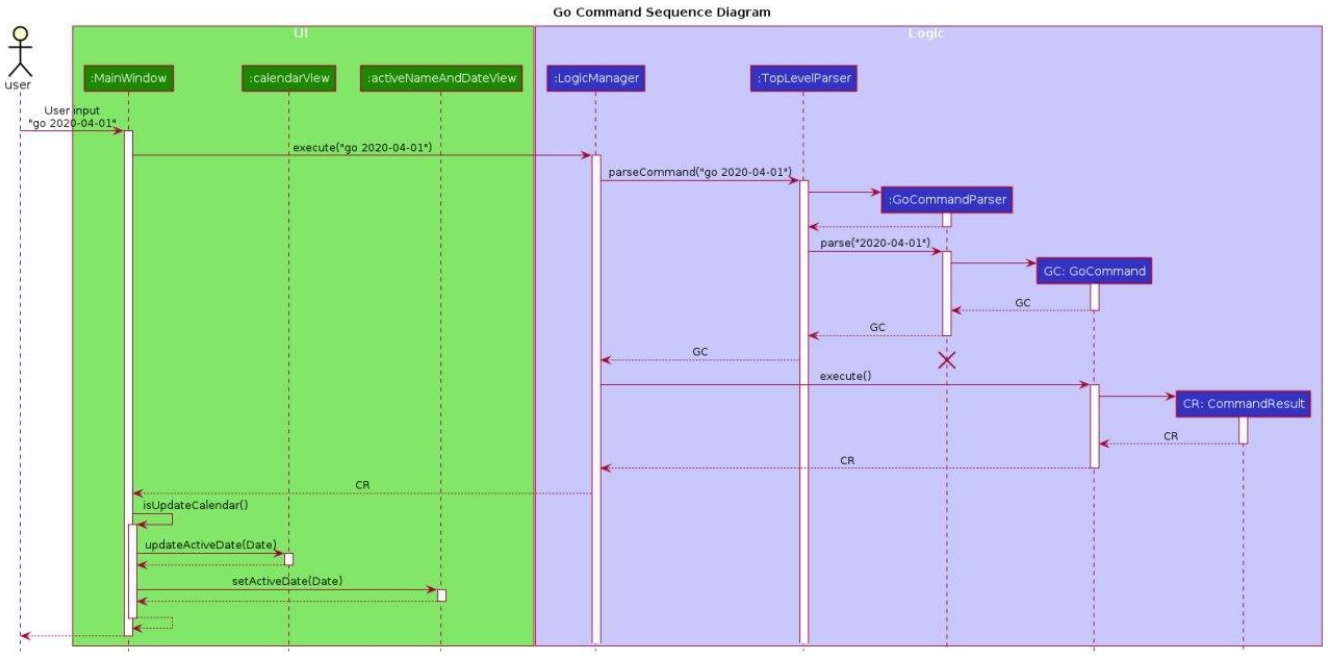

Figure 24. `UI` and `LOGIC` component for the `go 2020-04-01` command

Upon completion of the above command, the calendar view will update the active date to be `2020-04-01` and expenditures records for `2020-04-01` will be displayed.

If the user chooses to navigate through UI interaction with the calendar view (aka clicking on the date that is shown on the calendar). The implementation is very similar to the `go` command, `calendarView` will invoke `go` command when user click on the dates.

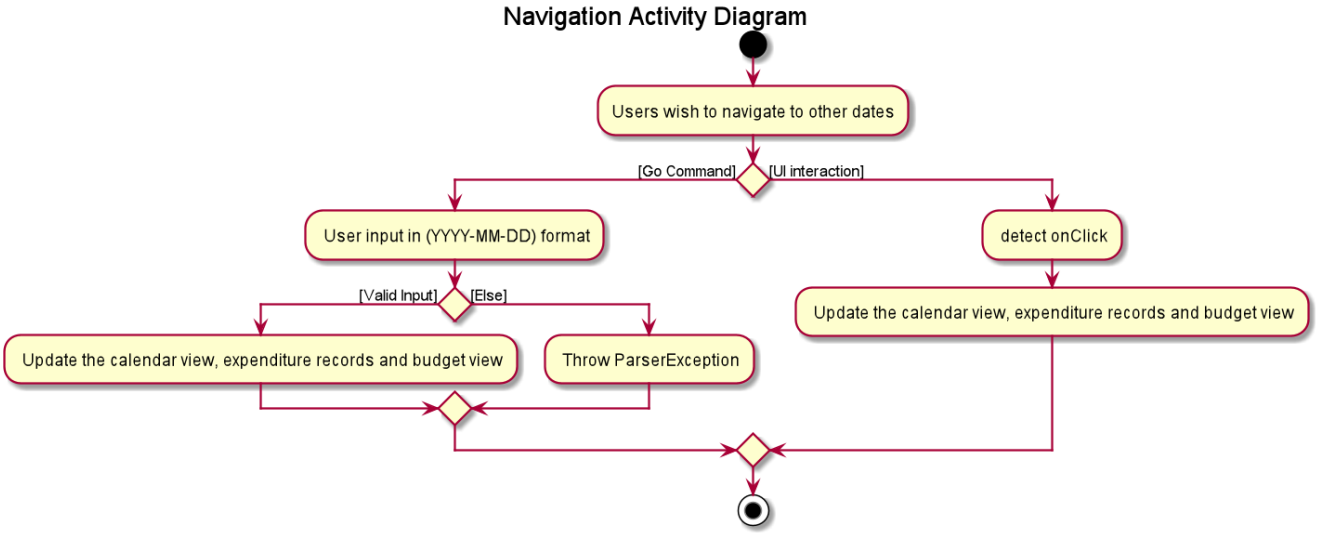

Figure 25. Activity diagram showing what happen when user wants to navigate to other date

4.6.3. Design Consideration

This section contains some of our design considerations for the calendar feature.

Consideration: How are we going to present the expenditure records.

Alternatives	Pros	Cons
1. Make use of a month list to contain all the expenditure records of the given month.	This is able to provide a concise view of expenditure view especially when there are only a small number of records.	This looks like excel sheet and users have to scroll all the way up if they want to view a date which is much earlier.
2. **[current choice]** Make use of a calendar view and only list out a given date's expenditure record. This automatically helps user to organize the records according to date.	User can make use of the calendar view to navigate between the dates, this is much more convenient than scrolling through a list. This helps user to organize the expenditure and keep it tidy. Especially helpful if there is lots of records.	It is much more troublesome to implement.

4.6.4. NOTE

Dates with negative year are allowed. E.g. `-1234-03-21` is allowed.
The developer team follows the range which is specified in the LocalDate API, released by Oracle.

4.7. Autocomplete feature (Lim Feng Yue)

Completes the command that the user is typing in the command box.

4.7.1. Rationale

The autocomplete feature makes it easier for user to know what commands there are in the application. As the application is also catered for users who prefer typing, this feature can be of great assistance and helps in efficiency.

4.7.2. Implementation

The autocomplete feature is facilitated by `AutoCompleteTextField`. It extends the `TextField` component of JavaFx and provides a dropdown of possible commands using `ContextMenu`.

Given below is an example usage scenario of the autocompletion.

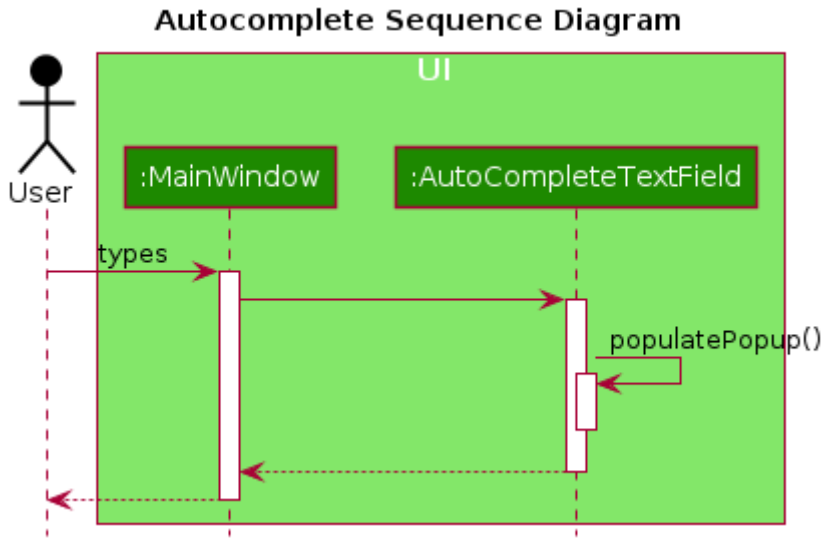

Figure 26. Autocomplete Sequence Diagram

1. Type into the command box. The function searches and filters potential commands that the user might use.
2. The commands will then be displayed in a dropdown format which the user can refer to when keying commands.

During the start up of the application, a list of commands are added to the `AutoCompleteTextField`. These commands will then be sorted lexicographically in the java implementation of `TreeSet`.

A `ChangeListener` is added the text field to 'listen' for changes in the input. `TreeSet#subSet()` is used to obtain all the commands between the previous text and the current text. For example, the textbox shows `ex` and a `p` is added, so the current text is `exp`. `TreeSet#subSet()` will obtain the commands that are lexicographically between `ex` and `exp`. These commands will then be shown in the autocomplete dropdown.

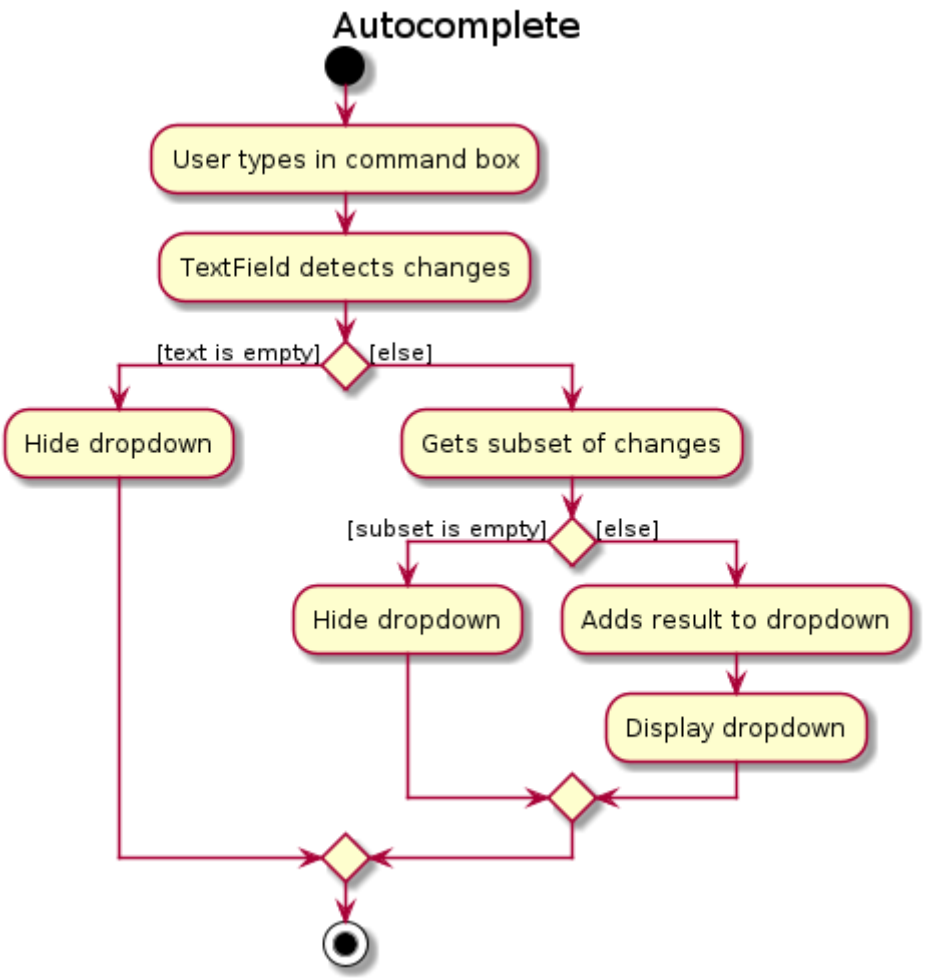

Figure 27. Autocomplete Activity Diagram

4.7.3. Design Consideration

Aspect: Usage of autocompletion

Alternatives	Pros	Cons
[current choice] Using up and down arrow keys to select autocompletion.	- Easy to implement.	- Requires the text field to be in focus.
Use of tab to simulate autocompletion like a terminal.	- Intuitive for people used to using a terminal.	- Will have to direct the tab keystroke to be used for autocompletion.

Aspect: Data structure

Alternatives	Pros	Cons
[current choice] Sorting lexicographically using `TreeSet`.	- Easy to implement.	- Can be slow if sorting through a huge number of strings.
Using a prefix trie.	- Extremely fast.	- Takes up a lot of space.

The current choice is chosen as the number of commands is not a lot, so high performance is not

required.

4.8. Configuration

Certain properties of the application can be controlled (e.g. user prefs file location, logging level) through the configuration file (default: `config.json`).

4.9. Logging

We are using `java.util.logging` package for logging. The `LogsCenter` class is used to manage the logging levels and logging destinations.

- The logging level can be controlled using the `logLevel` setting in the configuration file (See Section 4.8, "Configuration")
- The `Logger` for a class can be obtained using `LogsCenter.getLogger(Class)` which will log messages according to the specified logging level
- Currently log messages are output through: `Console` and to a `.log` file.

Logging Levels

- `SEVERE` : Critical problem detected which may possibly cause the termination of the application
- `WARNING` : Can continue, but with caution
- `INFO` : Information showing the noteworthy actions by the App
- `FINE` : Details that is not usually noteworthy but may be useful in debugging e.g. print the actual list instead of just its size

5. Documentation

Refer to the guide here.

6. Testing

Refer to the guide here.

7. Dev Ops

Refer to the guide here.

Appendix A: Product Scope

Target user profile:

- has a need to manage expenditure

- prefers desktop apps over other types
- can type fast
- prefers typing over mouse input
- is reasonably comfortable using CLI apps

Value proposition: manage expenditures faster than a typical mouse/GUI driven app & better organise them using disjoint accounts.

Appendix B: User Stories

Priorities: High (must have) - * * *, Medium (nice to have) - * *, Low (unlikely to have) - *

Priority	As a …	I want to …	So that I can…
* * *	new user	see usage instructions	refer to instructions when I forget how to use the App
* * *	lazy user	have an intuitive UI	spend less time navigating
* * *	multi-role user	have multiple disjoint accounts	use the app to track expenditure for different role
* * *	as a project director of my school club	create a partition between personal and project spending	keep track of personal spending as well as project spending, so that I can have an easier time keeping track of financial information
* * *	project leader	generate an expenditure report	document all the expenditure for future reference
* * *	busy and clumsy student	have a feature of undo and redo	recover my data from mistakes

Priority	As a …	I want to …	So that I can…
* * *	visual user	see the overview of my spending	have a clearer insight on my spending
* * *	time conscious user	take note of the time for each expenditure	plan my days to be in line with my spending
* * *	night owl	have a dark theme	protect my eyes at night
* *	disorganized user	categorize my expenditure	view my spending habit
* *	someone who is not mathematically inclined	have numbers that are intuitive	understand it easily
* *	user with many spending in the list	sort the expenditure	keep the expenditure organize

{More to be added}

Appendix C: Use Cases

(For all use cases below, the **System** is the `$AVE IT` and the **Actor** is the `user`, unless specified otherwise)

Use case: Acc Add

MSS

1. User requests to add a new account. Format: `acc add ACCOUNT`
2. `$AVE IT` will acknowledge and add this account into the list.

   Use case ends.

Extensions

1a. `$AVE IT` detects data is in wrong format.

- 1a1. `$AVE IT` will request for correct input format.
- Use case resumes from step 1.

1b. `$AVE It` detects a duplicate account name input.

- 1b1. `$AVE IT` will state that duplicate account name detected. Unable to add.
- Use case resumes from step 1.

Use case: Acc Delete

MSS

1. User requests to delete an existing account. Format: `acc delete ACCOUNT`
2. `$AVE IT` will acknowledge and delete this account from the list.

   Use case ends.

Extensions

1a. `$AVE IT` detects input is in wrong format.

- 1a1. `$AVE IT` will request for correct input format.
- Use case resumes from step 1.

1b. `$AVE IT` detects account name is non-existent.

- 1b1. `$AVE IT` will state that account is not found. Unable to delete.
- Use case resumes from step 1.

2a. User requests to deleted the only account in the list.

- 2a1. `$AVE IT` will create a default account to ensure there is at least an account in the list
- Use case ends.

2b. User requests to delete the account he is are viewing right now.

- 2b1. `$AVE IT` will checkout to a random existing account.
- Use case ends.

Use case: Acc Rename

MSS

1. User requests to rename an account. Format: `acc rename [OLD_NAME] NEW_NAME`
2. `$AVE IT` will acknowledge and state that the account has being renamed.

   Use case ends.

Extensions

1a. `$AVE IT` detects that input is in wrong format.

- 1a1. `$AVE IT` will request for correct input format.
- Use case resumes from step 1.

1b. `$AVE IT` detects that the account with `OLD_NAME` is non-existent.

- 1b1. `$AVE IT` will state that the account with the specified name was not found.
- Use case resumes from step 1.

1c. `$AVE IT` detects existence of account with `NEW_NAME`.

- 1c1. `$AVE IT` will state that a duplicate account was detected. Unable to add.
- Use case resumes from step 1.

Use case: Acc Checkout

MSS

1. User requests to check out another account. Format: `acc checkout ACCOUNT`
2. `$AVE IT` will acknowledge and state that the target account is checked out.

   Use case ends.

Extensions

1a. `$AVE IT` detects that input is in wrong format.

- 1a1. `$AVE IT` will request for correct input format.
- Use case resumes from step 1.

1b. `$AVE IT` detects that the account is non-existent.

- 1b1. `$AVE IT` will state that the account with the specified name was not found.
- Use case resumes from step 1.

Use case: Acc List

MSS

1. User requests to add a new account. Format: `acc list`
2. `$AVE IT` will acknowledge and state that the account has being renamed.

   Use case ends.

Extensions

1a. `$AVE IT` detects that input is in wrong format..

- 1a1. `$AVE IT` will request for correct input format.
- Use case resumes in step 1.

Use case: Acc Clear

MSS

1. User requests to clear all data in an account. Format: `acc clear`
2. `$AVE IT` will acknowledge and state that the account's data has been cleared.

   Use case ends.

Extensions

1a. `$AVE IT` detects that input is in wrong format.

- 1a1. `$AVE IT` will request for correct input format.
- Use case resumes in step 1.

Use case: Exp Add

MSS

1. User requests to add an expenditure record in the account which they are viewing right now. Format: `exp add -i INFO -a AMOUNT [-t TAG] [-d DATE]`
2. `$AVE IT` will acknowledge that a new expenditure has been added and show the details of the added expenditure.

   Use case ends.

Extensions

1a. `$AVE IT` detects that input is in wrong format.

- 1a1. `$AVE IT` will request for correct input format.
- Use case resumes from step 1.

1b. `$AVE IT` detects that the amount input is invalid.

- 1b1. `$AVE IT` will request for the amount to be a double.
- Use case resumes from step 1.

1c. `$AVE IT` detects that the tag is not specified.

- 1c1.`$AVE IT` will auto assign it to be `Others`
- 1c2.`$AVE IT` will acknowledge that a new expenditure has been added and show the details of the added expenditure.
- Use case ends.

1d. `$AVE IT` detects that date is not specified.

- 1d1. `$AVE IT` will add this expenditure record to the day which the calendar states.
- 1d2: `$AVE IT` will acknowledge that a new expenditure has been added and show the details of the added expenditure.
- Use case ends.

Use case: Exp Delete

MSS

1. User requests to delete an expenditure record in the account they are viewing right now.
   Format: `exp delete INDEX`
2. `$AVE IT` will acknowledge.

   Use case ends.

Extensions

1a. `$AVE IT` detects that input is in wrong format.

- 1a1. `$AVE IT` will request for correct input format.
- Use case resumes from step 1.

1b. `$AVE IT` detects that the index provided is invalid.

- 1b1. `$AVE IT` will state that the expenditure index provided is invalid.
- Use case resumes from step 1.

Use case: Exp Edit

MSS

1. User requests to edit an expenditure record in the account they are viewing right now.
   Format: `exp edit INDEX [-i INFO] [-a AMOUNT] [-t TAG] [-d DATE]`
2. `$AVE IT` will acknowledge, edit the relevant expenditure and the list will be auto sorted again.

   Use case ends.

Extensions

1a. `$AVE IT` detects that input is in wrong format.

- 1a1. `$AVE IT` will request for correct input format.
- Use case resumes in step 1.

1b. `$AVE IT` detects that the amount input is invalid.

- 1b1. `$AVE IT` will request for the amount to be a double.
- Use case resumes in step 1.

1c. `$AVE IT` detects that the date input is invalid.

- 1c1. `$AVE IT` WILL request a valid and non empty date.
- Use case resumes in step 1.

1d. `$AVE IT` detects that the index provided is invalid.

- 1d1. `$AVE IT` will state that the expenditure index provided is invalid.

- Use case resumes in step 1.

Use case: Repeat Add

MSS

1. User requests to add a repeating expenditure record in the account which they are viewing right now.
   Format: `repeat add -i INFO -a AMOUNT -sd START_DATE -ed END_DATE -p PERIOD [-t TAG]`
2. `$AVE IT` will acknowledge that a new repeat has been added and show the details of the added repeat.

   Use case ends.

Extensions

1a. `$AVE IT` detects that input is in wrong format.

- 1a1. `$AVE IT` will request for correct input format.
- Use case resumes from step 1.

1b. `$AVE IT` detects that the amount input is invalid.

- 1b1. `$AVE IT` will request for the amount to be a double.
- Use case resumes from step 1.

1c. `$AVE IT` detects that the start date or end date input is invalid.

- 1c1. `$AVE IT` will request a valid end date.
- Use case resumes from step 1.

1d. `$AVE IT` detects that the period input is invalid.

- 1d1. `$AVE IT` will request for the period to be `daily`, `weekly`, `monthly` or `annually`.
- Use case resumes from step 1.

1e. `$AVE IT` detects that the tag is not specified.

- 1e1. `$AVE IT` will auto assign it to be `Others`
- 1e2. `$AVE IT` will acknowledge that a new repeat has been added and show the details of the added repeat.
- Use case ends.

Use case: Repeat Delete

MSS

1. User requests to delete a repeating expenditure record from the account which they are viewing right now.
   Format: `repeat delete INDEX`

2. `$AVE IT` will acknowledge.

   Use case ends.

Extensions

1a. `$AVE IT` detects that input is in wrong format.

- 1a1. `$AVE IT` will request for correct input format'.
- Use case resumes from step 1.

1b. `$AVE IT` detects that the index input is invalid.

- 1b1. `$AVE IT` will state that the repeat index provided is invalid.
- Use case resumes from step 1.

Use case: Repeat Edit

MSS

1. User requests to edit a repeating expenditure record in the account which they are viewing right now.
   Format: `repeat edit INDEX [-i INFO] [-a AMOUNT] [-sd START_DATE] [-ed END_DATE] [-p PERIOD] [-t TAG]`
2. `$AVE IT` will acknowledge, edit the relevant `repeat` and the list will be auto sorted again.

   Use case ends.

Extensions

1a. `$AVE IT` detects that input is in wrong format.

- 1a1. `$AVE IT` will request for correct input format.
- Use case resumes from step 1.

1b. `$AVE IT` detects that the amount input is invalid.

- 1b1. `$AVE IT` will request for the amount to be a double.
- Use case resumes from step 1.

1c. `$AVE IT` detects that the start date or end date input is invalid.

- 1c1. `$AVE IT` will request a valid end date.
- Use case resumes from step 1.

1d. `$AVE IT` detects that the period input is invalid.

- 1d1. `$AVE IT` will request for the period to be `daily`, `weekly`, `monthly` and `annually`.
- Use case resumes from step 1.

1e. `$AVE IT` detects that the index input is invalid.

- 1e1. `$AVE IT` will state that the repeat index provided is invalid.

- 1e2. Back to step 1.
- Use case ends.

Use case: Report View

MSS

1. User requests to view a report of expenditure records in the account which they are viewing right now.
   Format: `report view -sd START_DATE -ed END_DATE -g GRAPH_TYPE -o ORGANISATION`
2. `$AVE IT` will acknowledge and pop up another window to show the relevant report.

   Use case ends.

Extensions

1a. `$AVE IT` detects that input is in wrong format.

- 1a1. `$AVE IT` will request for correct input format.
- Use case resumes from step 1.

1b. `$AVE IT` detects that the start date or end date input is invalid.

- 1b1. `$AVE IT` will request a valid start and end date.
- Use case resumes from step 1.

1c. `$AVE IT` detects that the graph type input is invalid.

- 1c1. `$AVE IT` will request a valid graph type.
- Use case resumes from step 1.
- 1d. `$AVE IT` detects that the organisation input is invalid.
- 1d1. `$AVE IT` will request a valid organisation.
- Use case resumes from step 1.

Use case: Report Export

MSS

1. User requests to export a report of expenditure records in the account which they are viewing right now.
   Format: `report export -sd START_DATE -ed END_DATE -g GRAPH_TYPE -o ORGANISATION -f FILE_NAME`
2. `$AVE IT` will acknowledge and export the report to the folder which has same location as `$AVE IT`.

   Use case ends.

Extensions

1a. Invalid command

- 1a1. `$AVE IT` will request for correct input format.
- 1a2. Back to step 1.
- use case end.

1b. Invalid start date or end date.

- 1b1. `$AVE IT` will request a valid start and end date.
- 1b2. Back to step 1.
- use case end.

1c. `$AVE IT` detects that the graph type input is invalid.

- 1c1. `$AVE IT` will request a valid graph type.
- Use case resumes from step 1.
- 1d. `$AVE IT` detects that the organisation input is invalid.
- 1d1. `$AVE IT` will request a valid organisation.
- Use case resumes from step 1.
- 1d. `$AVE IT` detects that file name input is invalid.
- 1d1. `$AVE IT` will request a valid file name.
- Use case resumes from step 1.

Use case: Generate report in report window

MSS

1. User request to generate new report.
2. `$AVE IT` will update report window to reflect input result.

   Use case ends.

Extensions

1a. `$AVE IT` detects that input is in wrong format.

- 1a1. `$AVE IT` will request for correct input format.
- Use case resumes from step 1.

1b. `$AVE IT` detects that the start date or end date input is invalid.

- 1b1. `$AVE IT` will request a valid start and end date.
- Use case resumes from step 1.

1c. `$AVE IT` detects that the graph type input is invalid.

- 1c1. `$AVE IT` will request a valid graph type.
- Use case resumes from step 1.

- 1d. `$AVE IT` detects that the organisation input is invalid.
- 1d1. `$AVE IT` will request a valid organisation.
- Use case resumes from step 1.

Use case: Set Budget

MSS

1. User requests to set a budget for a given month in the account which they are viewing right now.
   Format: `setbudget -a AMOUNT [-ym YEAR_MONTH]`
2. `$AVE IT` will acknowledge, budget view will be updated correspondingly.

   Use case ends.

Extensions

1a. `$AVE IT` detects that input is in wrong format.

- 1a1. `$AVE IT` will request for correct input format.
- Use case resumes in step 1.

1b. `$AVE IT` detects that the amount input is invalid.

- 1b1. `$AVE IT` will request for the amount to be a double.
- Use case resumes from step 1.

1c. `$AVE IT` detects that the date input is invalid.

- 1c1. `$AVE IT` WILL request a valid and non empty date.
- Use case resumes from step 1.

2a. `$AVE IT` detects that date is not specified.

- 2a1. `$AVE IT` will add this expenditure record to the day which the calendar states.
- 2a2. `$AVE IT` will acknowledge.
- use case end.

Use case: Find

MSS

1. User requests to find expenditure & repeat records with certain `keyword`(s) in the account which they are viewing right now.
   Format: `find [KEYWORD…] [-t TAG]`
2. `$AVE IT` will acknowledge and output a list of relevant records.

   Use case ends.

Extensions

1a. `$AVE IT` detects that input is in wrong format.

- 1a1. `$AVE IT` will request for correct input format.
- Use case resumes from step 1.

Use case: Go

MSS

1. User requests to view other date. Format: `go DATE`
2. `$AVE IT` will acknowledge and update the view.

   Use case ends.

Extensions

1a. `$AVE IT` detects that input is in wrong format.

- 1a1. `$AVE IT` will request for correct input format.
- Use case resumes from step 1.

1b. `$AVE IT` detects that the date input is invalid.

- 1b1. `$AVE IT` WILL request a valid and non empty date.
- Use case resumes from step 1.

Use case: Help

MSS

1. User requests for help.
2. `$AVE IT` will acknowledge and provide help.

   Use case ends.

Use case: Exit

MSS

1. User requests to exit.
2. `$AVE IT` will acknowledge and exit.

   Use case ends.

Appendix D: Non Functional Requirements

1. Should work on any mainstream OS as long as it has Java 11 or above installed.
2. Should be able to hold up to 1000 expenditures without a noticeable sluggishness in performance for typical usage.
3. Should be able to hold up to 100 accounts without a noticeable sluggishness in performance for typical usage.
4. A user with above average typing speed for regular English text (i.e. not code, not system admin commands) should be able to accomplish most of the tasks faster using commands than using the mouse.

Appendix E: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X

Command Line Input

Command line interface (CLI) is a text-based interface that is used to operate software and operating systems while allowing the user to respond to visual prompts by typing single commands into the interface and receiving a reply in the same way.

Appendix F: Instructions for Manual Testing

Given below are instructions to test the app manually.

NOTE	These instructions only provide a starting point for testers to work on; testers are expected to do more *exploratory* testing. Only the last valid prefix will be taken into account. E.g. exp add -i chicken rice -a 3.5 -a 4.0. The new expenditure record will have an amount fo 4.00 instead of 3.50. **This applies to all other command**.

F.1. Launch and Shutdown

F.1.1. Initial launch

a. Download the jar file and copy into an empty folder

b. Double-click the jar file
Expected: Shows the GUI with a set of sample expenditures & repeats in a few accounts. The window size may not be optimum.

c. If you are running the jar file through cmd, the data, log and .json files will be located at the directory where you run the jar file.

F.1.2. Saving window preferences

a. Resize the window to an optimum size. Move the window to a different location. Close the window.

b. Re-launch the app by double-clicking the jar file.
Expected: The most recent window size and location is retained.

F.2. Account command test

F.2.1. Add a new account

a. Prerequisites:
The new `ACCOUNT_NAME` must not exist in the system.
Name must be a word (contains no space).

b. Test case: `acc add projectXYZ`
Expected: A new account named `projectXYZ` is added to the account list.

F.2.2. View all accounts

Test case: `acc list`
Expected: All the accounts in the application will be displayed.

F.2.3. Rename account

a. Prerequisites:
The account whose name you want to change, must exist in the system.
The new name must not be an existing account's name.

b. Test case: `acc rename projectXYZ projectABC`
Expected: the account's name with project-xyz change to project-abc.

F.2.4. Checkout an account

a. Prerequisites:
The account which you want to checkout to, must exist in the system.

b. Test case `acc checkout personal`
Expected: A response will be given to denote a change in account.

F.2.5. Delete account

a. Prerequisites:
The account which you want to delete, must exist in the system.

b. Test case: `acc delete projectABC`
Expected: the account will be deleted.

```
Note:
If that is the only account, a new default account will be auto generated.
If you deleted the account which you are viewing on, you will be auto checked-out
to another account after deletion.
```

F.2.6. Clear data of an account

a. Prerequisites:
   The active account will be the account whose data will be cleared.

b. Test case: `acc clear`
   Expected: the current account's data will be all cleared.

F.3. Expenditure command test

F.3.1. Add a new expenditure

a. Prerequisites:
   The command input must be in the right format.

b. Test case: `exp add -i Chicken rice -a 3.50 -t Lunch`
   Expected: New expenditure added: Chicken rice Amount: 3.50 Date: 2020-04-12 Tag: lunch. This will be added to current date (For this test case: current date is set to 2020-04-12").

c. Test case: `exp add -i Chicken rice -a 3.50 -t Lunch -d 2020-04-01`
   Expected: New expenditure added: Chicken rice Amount: 3.50 Date: 2020-04-01 Tag: lunch.

F.3.2. Delete an expenditure

a. Prerequisites:
   The index provided must be valid and it is an `expenditure` record.

b. Test case: `exp delete 1`
   Expected: Deleted Expenditure: Chicken rice Amount: 3.50 Date: 2020-04-12 Tag: lunch.

F.3.3. Edit an expenditure

a. Prerequisites:
   The command input must be valid, index provided must refer to an `expenditure` record.

b. Test case: `exp edit 1 -t meal`
   Expected: Edited Expenditure: Chicken rice Amount: 3.50 Date: 2020-04-01 Tag: meal.

F.3.4. Expenditure list

a. Test case: `exp list`
   Expected: View all the expenditure records.

```
Note:
This is mainly used to exit the find mode.
```

F.4. Repeat command test

F.4.1. Add a new repeat expenditure

a. Prerequisites:
The command input must be in the right format.
The end date must be equal or after the start date.

b. Test case: `repeat add -i bus fare -a 1.50 -sd 2020-01-01 -ed 2021-01-01 -p daily -t Transport`
Expected: New repeat added: bus fare Amount: 1.50 Start Date: 2020-01-01 End Date: 2021-01-01 Interval: daily Tags: Transport. will be added to current date (For this test case: current date is set to 2020-04-12").

F.4.2. Delete a repeat

a. Prerequisites:
The index provided must be valid and it is an `repeat` record.

b. Test case: `repeat delete 1` Expected: Deleted Repeat: bus fare Amount: 1.50 Start Date: 2020-01-01 End Date: 2021-01-01 Interval: daily Tags: Transport.

F.4.3. Edit a repeat

a. Prerequisites:
The command input must be valid, index provided must be refer to an `repeat` record.

b. Test case: `repeat edit 2 -sd 2020-04-01 -ed 2020-05-01`
Expected: Edited Repeat: bus fare Amount: 1.50 Start Date: 2020-04-01 End Date: 2020-05-01 Interval: daily Tags: Transport.

F.5. Report command test

F.5.1. View report

Main Window

a. Prerequisites:
The input parameters must be valid.

b. Test case: `report view -sd 2020-04-01 -ed 2020-05-31 -g pie -o month`
Expected: Pie chart with months where spending is not zero will be shown on the report window. If all months have zero dollar spending, then an empty Pie Chart will be created.

Report Window

a. Prerequisites:
The input parameters must be valid.

b. Test case: `view 2020-04-01 2020-05-31 bar tag`
Expected: Bar chart with spending categorised by tags will be shown on the report window. If there are no tags, then an empty Bar chart will be shown.

F.5.2. Export report

Main Window

a. Prerequisites:
The input parameters must be valid.

b. Test case: `report export -sd 2020-04-01 -ed 2020-05-31 -g bar -o tag -f filename`
Expected: Bar chart with spending categorised by tags will be exported to file with filename. If there are no tags, then an empty Bar chart will be exported.

Report Window

a. Prerequisites:
There is a report generated and input parameters must be valid.

b. Test case: `export filename`
Expected: Exports the current graph shown in report window to file named filename.

F.5.3. Print report

Main Window

a. Prerequisites:
There is a printer set up and input parameters must be valid.

b. Test case: `report print -sd 2020-04-01 -ed 2020-05-31 -g pie -o month`
Expected: Pie chart with months where spending is not zero will be printed. If all months have zero dollar spending, then an empty Pie Chart will be printed.

Report Window

a. Prerequisites:
There is a report generated and a printer set up.

b. Test case: `print`
Expected: The current graph shown in report window will be printed.

F.6. General command test

F.6.1. Go

a. Prerequisites:

The date input must be valid.

b. Test case: `go 2020-03-11`
Expected: We are at : 2020-03-11.

F.6.2. Set Budget

a. Test case: `setbudget -a 1000`
Assuming the active date is `2020-04-01`
Expected: The budget for the April 2020 will be set to 1000. The budget view will be updated with correspond values.

b. Test case: `setbudget -a 1000 -ym 2020-01`
Expected: The budget for the January 2020 will be set to 1000. The budget view will be updated with correspond values.

F.6.3. Find

a. Test case: 'find rice'
Expected: Will enter find mode: Expenditure and repeat records which contain `rice` will be shown.

b. Test case: 'find rice -t Food'
Expected: Will enter find mode: Expenditure and repeat records which contain `rice` and the tag `Food` will be shown.

c. Test case: 'find -t Food'
Expected: Will enter find mode: Expenditure and repeat records which contain the tag `Food` will be shown.

```
Note:
Users have to make use of "exp list" to exit find mode.
```

F.6.4. Exit

a. Test case: `exit`
Expected: `$AVEIT` will terminate.

Appendix G: Effort (Ng Xinpei, Zheng Shaopeng)

To date, our application has about 13kLOC. This meant that much coordination was needed between all of the team members. The COVID-19 situation has made communication tougher, however we managed to do online Skype calls routinely which improved coordination efforts.

G.1. Entities

AB3 contains and deals mainly with the Person entity only. However, in $AVE IT, there are multiple entities such as Expenditure, Repeat Account and Report that are required. While we managed to refactor some code from AB3 , for example changing Person from AB3 to Expenditure in $AVE It we had to create the other entities from scratch.

G.2. UI

AB3 only has one list panel view whereas $AVE IT boosts a calendar view and budget view on top of the list panel. Furthermore, an additional pop out window, Report Window is also added. This meant that much work was required to change AB3 's UI to the current UI we have. On top of that, we also needed in new UI components that are not present in AB3 such as the calendar UI. Our UI is time sensitive ,meaning that when users open the app it will automatically show today's date as well list panel of expenditures. This made implementation more challenging as we had to ensure that our UI responds to changes in timing.

G.3. Features

On top of AB3 's feature such as adding and editing an entity, we have implemented additional features such as acc checkout , report export and go today.

G.4. Others

Throughout the whole process, the team has ensured that we maintained high level of code quality through the usage of Continuous Integration(CI) tools such as Travis.