Financial Ghost - Developer Guide

By: CS2113T-W12-2 Since: September 2019

1. Setting Up

This section will guide you in setting up the Financial Ghost project in your local machine for the first time.

1.1. Prerequisites

1. JDK 11 or later
2. IntelliJ IDE: IntelliJ by default has Gradle and JavaFx plugins installed. Do not disable them. If you have disabled them, go to File > Settings > Plugins to re-enable them.

1.2. Setting up the project in your computer

1. Fork this repo, and clone the fork to your computer
2. Open IntelliJ (if you are not in the welcome screen, click File > Close Project to close the existing project dialog first)
3. Set up the correct JDK version for Gradle
    - Click Configure > Project Defaults > Project Structure
    - Click New… and find the directory of the JDK
4. Click Import Project
5. Locate the build.gradle file and select it. Click OK
6. Click Open as Project
7. Click OK to accept the default settings
8. Open a console and run the command gradlew run (Mac/Linux: ./gradlew run). It should finish with the BUILD SUCCESSFUL message.
   This will generate all resources required by the application and tests.

1.3. Verifying the setup

1. Run the MainApp and try a few commands
2. Run the tests to ensure they all pass.

1.4. Configurations to do before writing code

1.4.1. Configuring the Coding Style

This project follows java coding standards. IntelliJ's default style is mostly compliant with ours but it uses a different import order from ours. To rectify,

1. Go to `File > Settings…` (Windows/Linux), or `IntelliJ IDEA` > `Preferences…` (macOS)
2. Select `Editor` > `Code Style` > `Java`
3. Click on the `Imports` tab to set the order
    a. For `Class count to use import with '*'` and `Names count to use static import with '*'`: Set to `999` to prevent IntelliJ from contracting the import statements
    b. For `Import Layout`: The order is `import static all other imports`, `import java.*`, `import javax.*`, `import org.*`, `import com.*`, `import all other imports`. Add a `<blank line>` between each `import`
4. Optionally, you can follow the gradleTutorial.md document to configure Intellij to check style-compliance as you write code.

1.4.2. Updating documentation to match your fork

After forking the repo, links in the documentation will still point to the `AY1920S1-CS2113T-W12-2/main` repo. If you plan to develop this as a separate product (i.e. instead of contributing to the `AY1920S1-CS2113T-W12-2/main`) , you should replace the URL in the variable `repoURL` in `DeveloperGuide.pdf` and `UserGuide.pdf` with the URL of your fork.

1.4.3. Setting up CI

Set up Travis to perform Continuous Integration (CI) for your fork. See **UsingTravis.adoc** to learn how to set it up.

After setting up Travis, you can optionally set up coverage reporting for your team fork (see **UsingCoveralls.adoc**).

Coverage reporting could be useful for a team repository that hosts the final version but it is not that useful for your personal fork.

Optionally, you can set up AppVeyor as a second CI (see **UsingAppVeyor.adoc**).

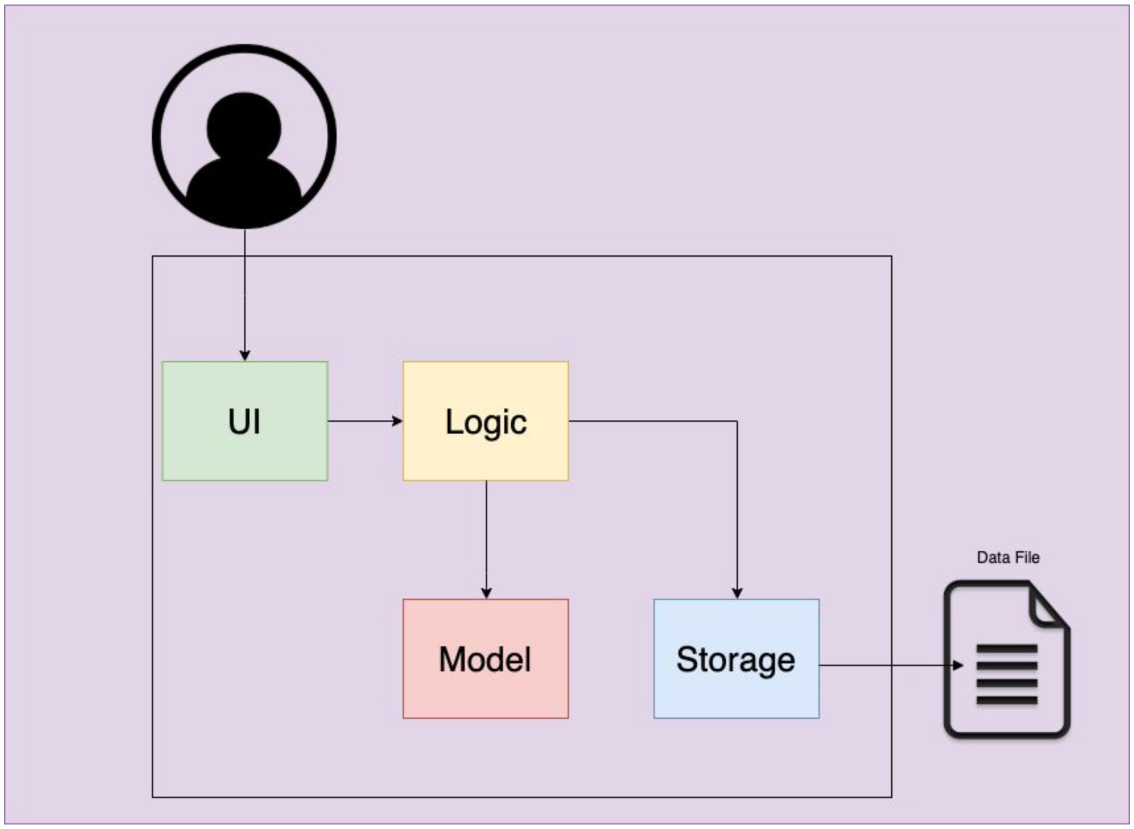
Having both Travis and AppVeyor ensures your App works on both Unix-based platforms and Windows-based platforms (Travis is Unix-based and AppVeyor is Windows-based).

1.4.4. Getting started with coding

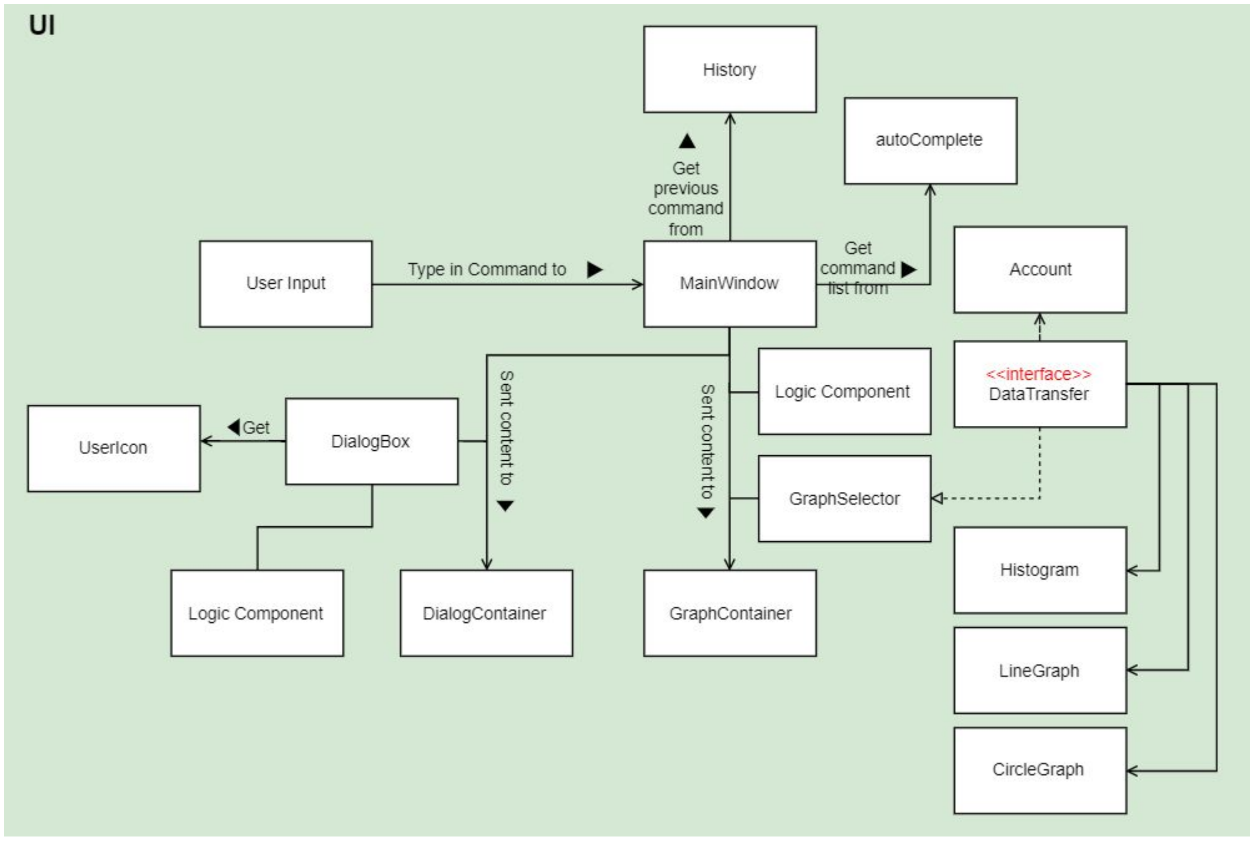
When you are ready to start coding, we recommend that you get some sense of the overall design by reading about Financial Ghost’s architecture.

2. Design

2.1. Architecture



2.2. UI Component

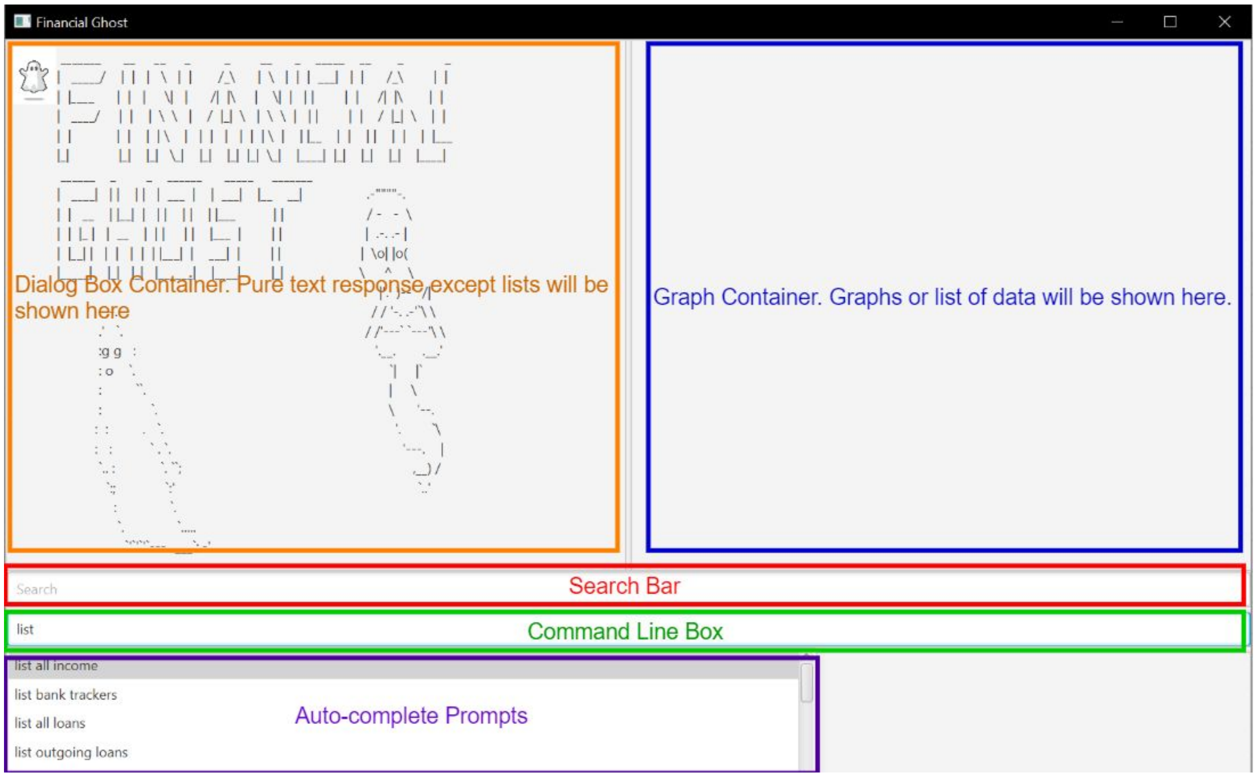


UI component is launched by launcher.java and it invokes Main class and MainWindow class to manipulate the overall GUI presented to the user for he/she to type in commands. Based on different user input, the MainWindow class will respond in different ways.

1. The text response only. This type of input will be handled by handleUserInput method. In handleUserInput, class Duke will be invoked and proceed to logic part get the corresponding output string presented to the user. When adding the message into the dialog box, UserIcon is also initialized and added into the dialog box.

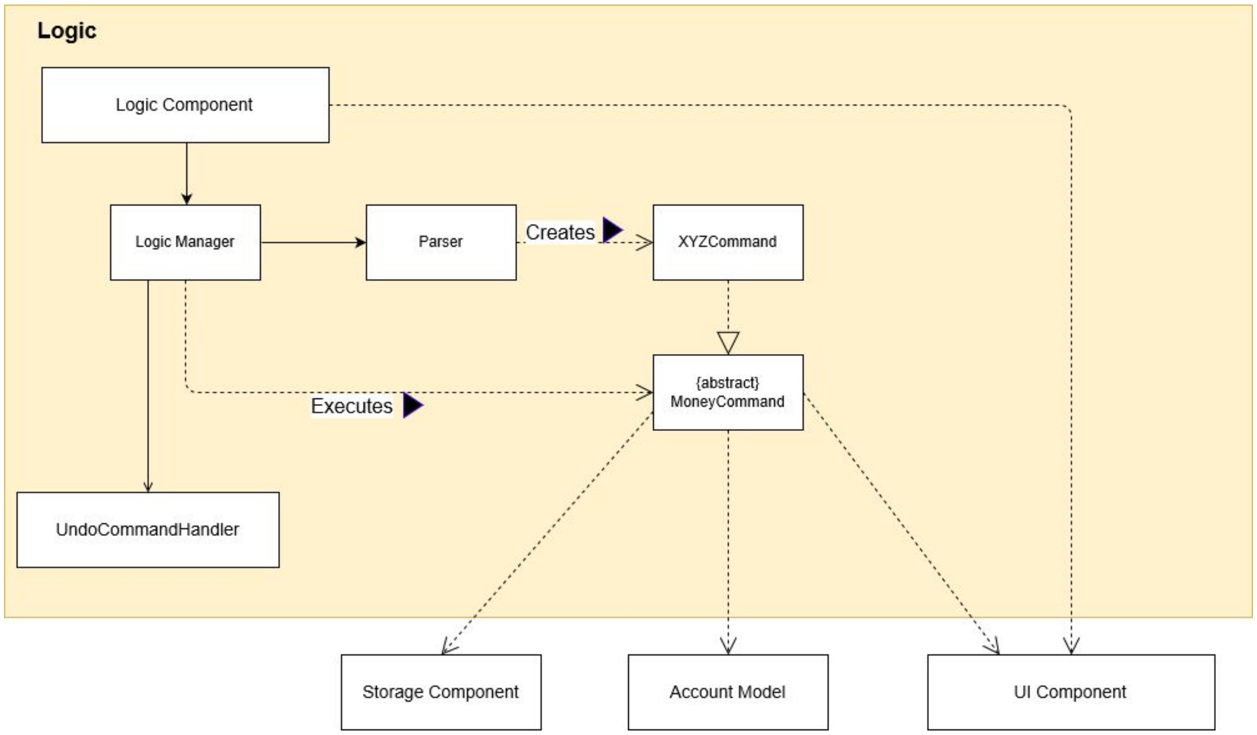
2. The search bar. This type is implemented as a search bar above the main command input box. Type in the keywords which will be handled by handleSearchInput method in MainWindow class then the relevant information including income, expenditure and loans etc. will be shown on the right hand side pane.

3. Graph and text response. This type of commands mainly require FG for a graph in order to get direct illustration about the current financial statistics. Graphs will be handled in GraphSelector initialized by MainWindow class and the text confirmation will be sent by Duke.



Moreover, the feature “auto-complete” is implemented in order to make typing command more user-friendly. When user starts to type in the command, a box of possible commands will be shown below the command line input box. This auto-complete function is handled by autoCompleteFunction in MainWindow class which gets the list from autoComplete class. In addition, “History” feature is implemented which allow user to use arrow button to go back to their previous commands. “History” feature is managed by help method which invokes getPreviousCommand method in History class to get previous command.

2.3. Logic Component



1. The Logic component uses the Parser class to parse the user's command from the UI.

2. The user's input is processed by the moneyParse method from the Parser class.

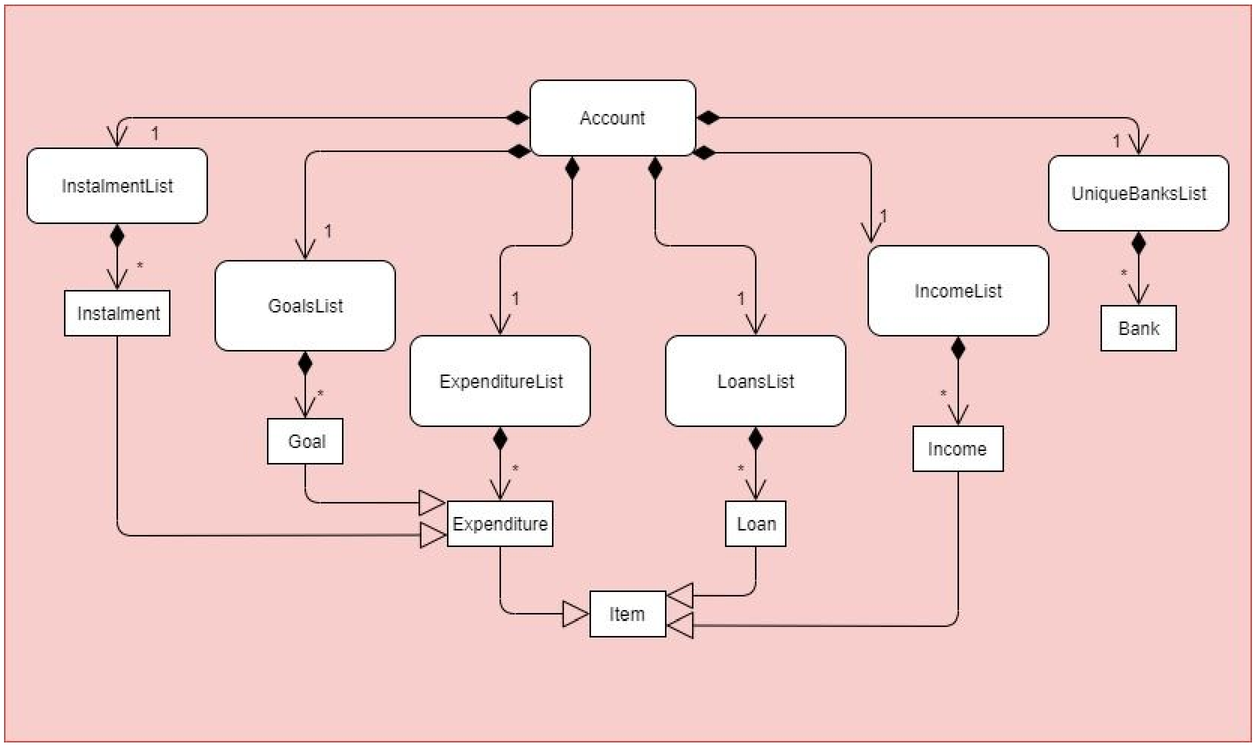
3. This results in a MoneyCommand object which is executed by the LogicManager (FG).

4. Upon execution of the MoneyCommand, it will affect the Account model or trigger an output in UI.

5. The result of the command execution will be stored in the ui object as outputString and graphContainerString.The ui object is passed back to UI as a String to be displayed to the user.

2.4. Model Component

Structure of the model component is shown in the below diagram.



API: Account.java

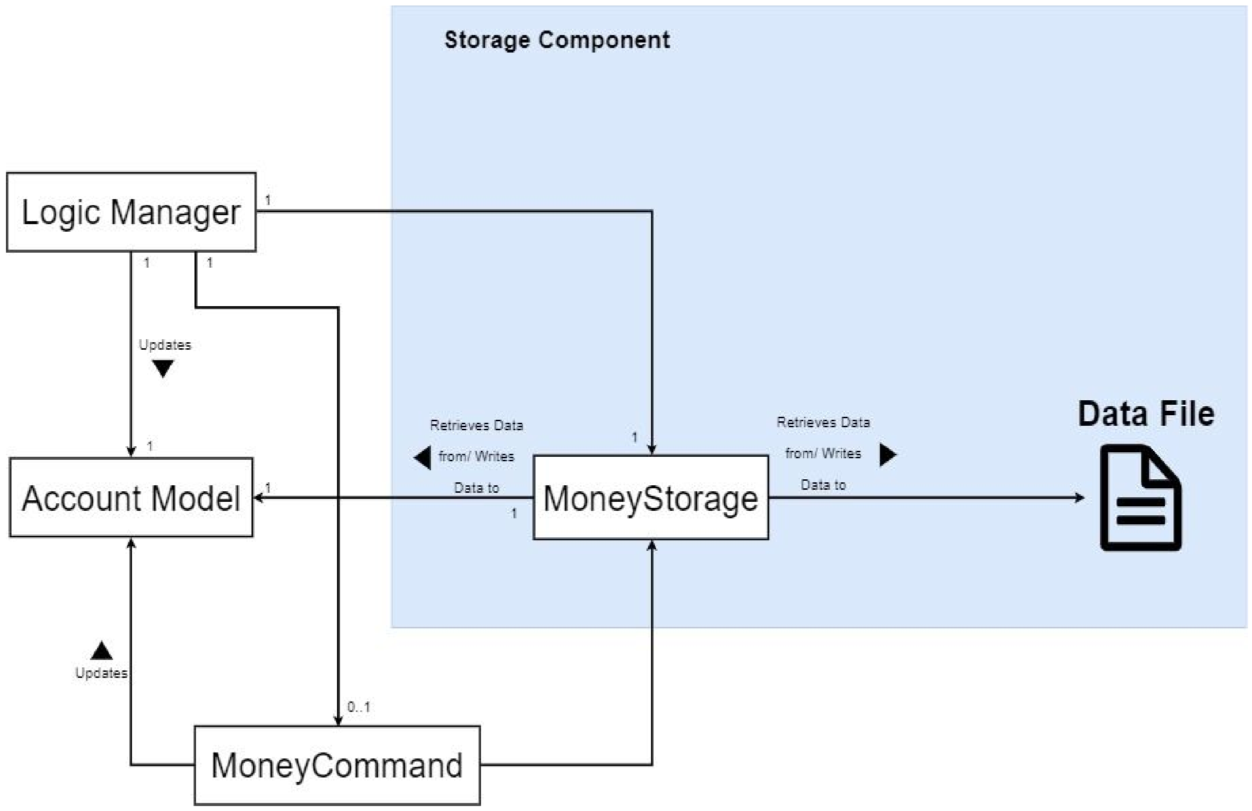
The model,

- Stores the Financial Ghost User Data.
- Does not depend on any of the other three components.

2.5. Storage Component

This section describes how the program saves information keyed into Financial Ghost through into the hard disk.

The structure diagram of the Storage Component can be found below.

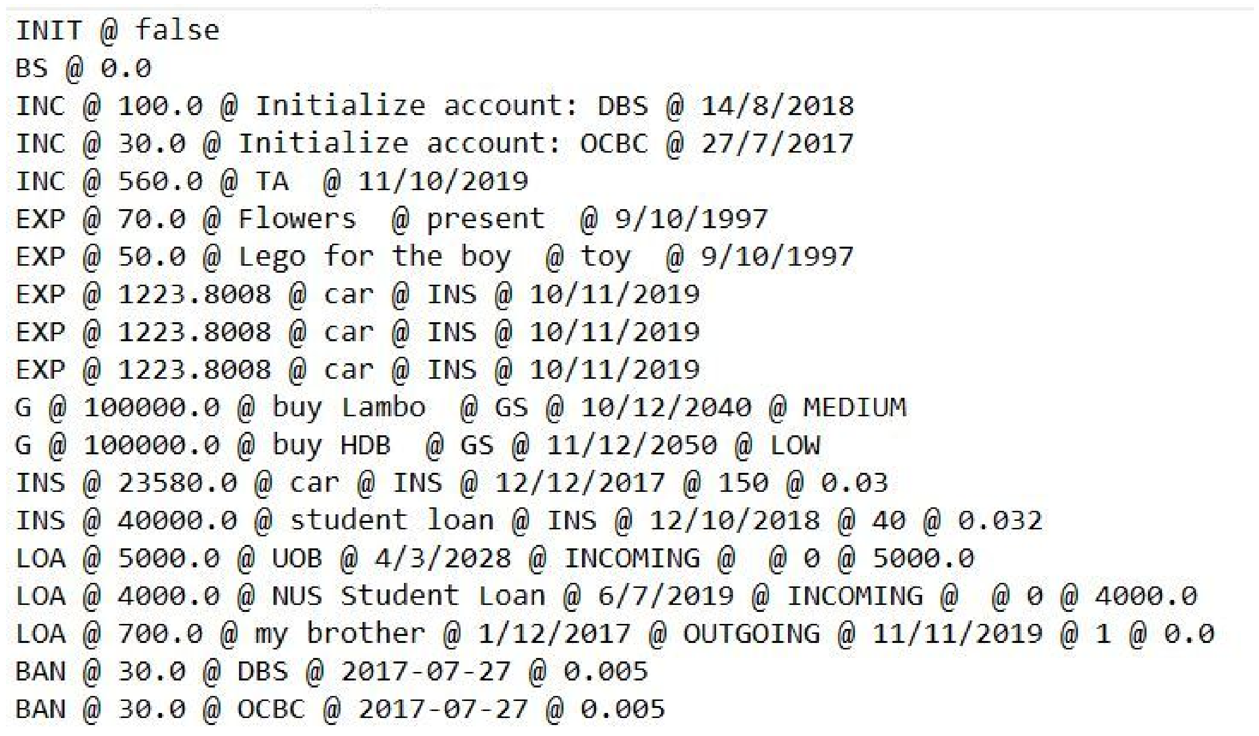


● All the user's information is stored in a .txt file generated by the program upon initialisation.

● The .txt file is written after every command is issued, and written once more when the user exits the program.

● At the start of every session, the class MoneyStorage reads and parses the information written in the .txt file and loads it into the Account Model.

A sample of a user's account .txt file is displayed in the screenshot below.



```
INIT @ false
BS @ 0.0
INC @ 100.0 @ Initialize account: DBS @ 14/8/2018
INC @ 30.0 @ Initialize account: OCBC @ 27/7/2017
INC @ 560.0 @ TA  @ 11/10/2019
EXP @ 70.0 @ Flowers  @ present  @ 9/10/1997
EXP @ 50.0 @ Lego for the boy  @ toy  @ 9/10/1997
EXP @ 1223.8008 @ car @ INS @ 10/11/2019
EXP @ 1223.8008 @ car @ INS @ 10/11/2019
EXP @ 1223.8008 @ car @ INS @ 10/11/2019
G @ 100000.0 @ buy Lambo  @ GS @ 10/12/2040 @ MEDIUM
G @ 100000.0 @ buy HDB  @ GS @ 11/12/2050 @ LOW
INS @ 23580.0 @ car @ INS @ 12/12/2017 @ 150 @ 0.03
INS @ 40000.0 @ student loan @ INS @ 12/10/2018 @ 40 @ 0.032
LOA @ 5000.0 @ UOB @ 4/3/2028 @ INCOMING @  @ 0 @ 5000.0
LOA @ 4000.0 @ NUS Student Loan @ 6/7/2019 @ INCOMING @  @ 0 @ 4000.0
LOA @ 700.0 @ my brother @ 1/12/2017 @ OUTGOING @ 11/11/2019 @ 1 @ 0.0
BAN @ 30.0 @ DBS @ 2017-07-27 @ 0.005
BAN @ 30.0 @ OCBC @ 2017-07-27 @ 0.005
```

The data is categorised by headers and separated by “@” characters. The table below shows a breakdown of what the data represents.

Data Type	**Header**	**Format**	**Remarks**
Initialisation	INIT	INIT @ true/false	Checks if account is a new user
Base Savings	BS	BS @ [amount]	Records base-savings of user
Income	INC	INC @ [amount] @ [description] @ [date of payday]	Records user income
Expenditure	EXP	EXP @ [amount] @ [description] @ [category] @ [date spent]	Records user expenditure
Goals	G	G @ [amount] @ [description] @ [goal type] @ [date to reach goal] @ [priority level]	Records user’s goals
Instalments	INS	INS @ [amount] @ [description] @ [category] @ [date spent] @ [number of payments to be made]	Records the instalments the user

		@ [annual interest rate]	has to pay
Loans	LOA	LOA @ [amount] @ [description] @ [date of loan] @ INCOMING/OUTGOING @ [date settled] @ 0/1 @ [outstanding amount]	Records incoming and outgoing loans to the user. The field with 0 or 1 indicates if the loan has been settled or not (0 = Not settled, 1 = settled)
Bank Account	BAN	BAN @ [bank name] @ [initial balance] @ [date of entry] @ [annual interest rate]	Records bank accounts of user

3. Implementation

This section describes some noteworthy details on how certain features are implemented.

3.1. Money In/Out Feature

The base feature of Financial Ghost.
It allows users to track their monetary inflow/outflow in form of income and expenditure.

It comprises of the following functions:

1. Add Income
2. Add Expenditure
3. List Income
4. List Expenditure
5. Delete Income
6. Delete Expenditure

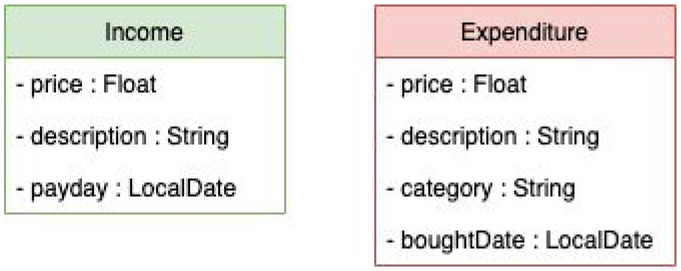
3.1.1. Add Commands Function (Add Income & Add Expenditure)

The add commands for money in/out are facilitated by the `AddIncomeCommand` and `AddExpenditureCommand` from the `Logic` Component.

For `AddIncomeCommand`, an `Income` object is instantiated and added into the `incomeListTotal` (ArrayList of Income) in the `Model` component.

For `AddExpenditureCommand`, an `Expenditure` object is instantiated and added into the `ExpListTotal` (ArrayList of Expenditure) in the `Model` component.

The following diagram illustrates the Income and Expenditure objects



Given below is an example usage scenario of how the add command behaves at each step, with variations depending on whether it is an `AddIncomeCommand` or an `AddExpenditureCommand`.

Step 1: The user calls the add command from the GUI with its respective parameters
e.g. add income TA /amt 500 /payday 1/10/2019 (To add income source)
e.g. spent chicken rice /amt 4.50 /cat food /on 4/10/2019 (To add an expenditure)

Step 2: The LogicManager calls moneyParse on Parser with the user input.

Step 3: Parser is called and it constructs either an `AddIncomeCommand` object or an `AddExpenditureCommand` object depending on user input . The Object is returned to LogicManager.

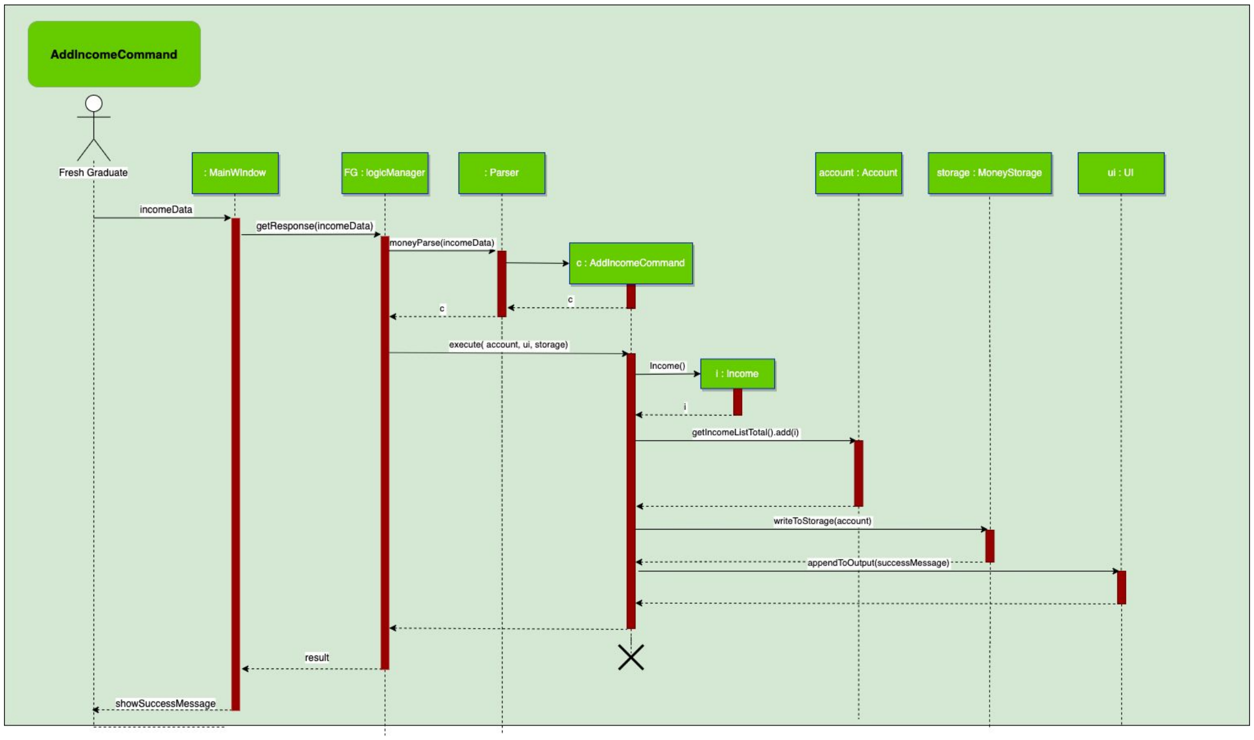
Step 4: The LogicManager calls execute() on the `AddIncomeCommand/AddExpenditureCommand` object.

Step 5: The Logic Component will interact with Account.

💵	`AddIncomeCommand`: Constructs an `Income` object to be added to IncomeListTotal in Account
💵	`AddExpenditureCommand`: Constructs an `Expenditure` object to be added to ExpListTotal in Account

Step 6: `AddIncomeCommand/AddExpenditureCommand` will call appendToOutput a Success Message.

The following sequence diagram illustrates how the `AddIncomeCommand` is carried out



NOTE: `AddExpenditureCommand` follows the same sequence, albeit constructing an `Expenditure` object and adding it to `ExpListTotal` with the method `getExpListTotal`

3.1.2. List Command Function (List Income & List Expenditure)

The list command is facilitated by the `ListIncomeCommand` and ListExpenditureCommand from the Logic component.

Based on user input, `Logic` will either execute `ListIncomeCommand` to list all `Income` objects in `incomeListTotal` (ArrayList of `Income`) or `ListExpenditureCommand` to list all `Expenditure` objects in `expListTotal` (ArrayList of `Expenditure`) in the `Model` component.

Step 1. The user calls either `ListIncomeCommand` or `ListExpenditureCommand` with the relevant input

- list all income (`ListIncomeCommand`)
- list all expenditure (`ListExpenditureCommand`)

Step 2. LogicManager calls MoneyParse command with user input.

Step 3. Paser is called and returns the corresponding list command (`ListIncomeCommand`/`ListExpenditureCommand`) object to LogicManager.

Step 4. The LogicManager calls method execute() on the object.

Step 5. Depending on the command called:

🗒	`ListIncomeCommand`: Calls a method on the `Model` component to return `incomeListTotal` and call `appendToGraphContainer` the ArrayList of `Income`. Calls `appendToOutput` a Success Message
🗒	`ListExpenditureCommand`: Calls a method on the `Model` component to return `expListTotal` and call `appendToGraphContainer` the ArrayList of `Expenditure`. Calls `appendToOutput` a Success Message

Step 6. The command will return the Success Message and the corresponding List which will be displayed in the GUI.

3.1.3. Delete Command Function (Delete Income & Delete Expenditure)

The delete commands for money in/out are facilitated by the `DeleteIncomeCommand` and Delete`ExpenditureCommand` from the `Logic` Component.

For `DeleteIncomeCommand`, an `Income` object within the `incomeListTotal` (ArrayList of Income) in the `Model` component is removed according to the serial number provided in the user input.

For `DeleteExpenditureCommand`, an `Expenditure` object within the `ExpListTotal` (ArrayList of Expenditure) in the `Model` component is removed according to the serial number provided in the user input.

Given below is an example usage scenario of how the delete command behaves at each step, with variations depending on whether it is a `DeleteIncomeCommand` or a `DeleteExpenditureCommand`.

Step 1: The user calls the delete command from the GUI with the serial number of the object within the List to be deleted.
e.g. delete income 2 (To delete an income source)
e.g. delete expenditure 7 (To delete an expenditure)

Step 2: The LogicManager calls moneyParse on Parser with the user input.

Step 3: Parser is called and it constructs either a `DeleteIncomeCommand` object or a `DeleteExpenditureCommand` object depending on user input . The Object is returned to LogicManager.

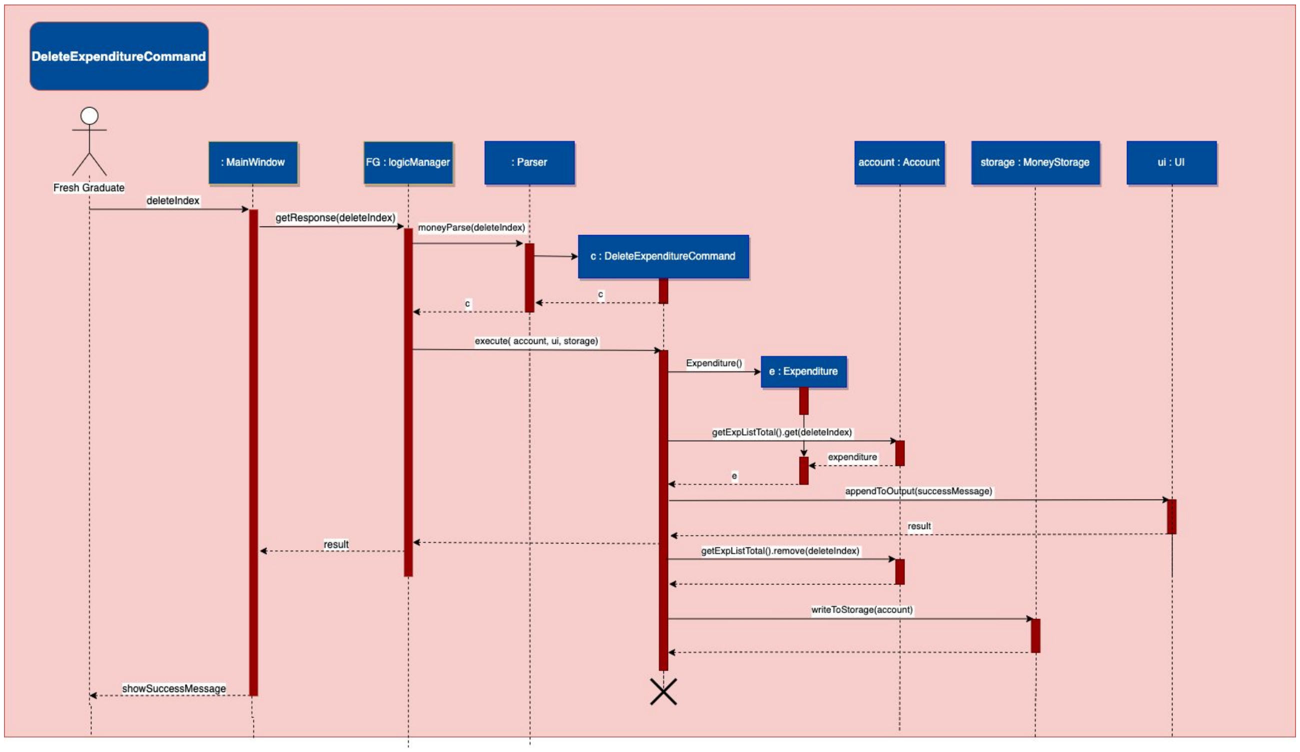
Step 4: The LogicManager calls execute() on the `DeleteIncomeCommand/DeleteExpenditureCommand` object.

Step 5: The Logic Component will interact with Account.

🗑	`AddIncomeCommand`: Calls a method on the `Model` component to return `incomeListTotal` and calls `remove(serialNo)` on the `Income` object to remove it from `incomeListTotal`
🗑	`AddExpenditureCommand`: Calls a method on the `Model` component to return `expListTotal` and calls `remove(serialNo)` on the `Expenditure` object to remove it from `expListTotal`

Step 6: `DeleteIncomeCommand/DeleteExpenditureCommand` will call appendToOutput a Success Message.

The following sequence diagram illustrates how the `DeleteExpenditureCommand` is carried out



NOTE: `DeleteIncomeCommand` follows the same sequence, albeit constructing an `Income` object and removing it from `IncomeListTotal` with the method `getIncomeListTotal`

3.1.4. Design Considerations

As the base function of our programme, Financial Ghost, there were many design considerations that arose during development.

- Alternative 1: Use of a single class, called Item, to serve as objects to record both money inflow and outflow
  - Pros: Less classes and simpler implementation
  - Cons: Income and expenditure required different attributes
- Alternative 2 (current choice): Create separate classes for recording income and expenditure which extend Item
  - Pros: Allows income and expenditure to take unique attributes
  - Cons: Separate ArrayLists implemented in Model to store either Income or Expenditure Objects

For List commands, we intended to allow users to choose between listing all money in/out or just the money in/out for the month. The development of this function lead to the creation of the Archive function.

- Alternative 1: Create another 2 ArrayLists to record the money in/out for the month (`IncomeListMonthly` & `ExpListMonthly`)
  - Pros: Easy to implement
  - Cons: Abundance of duplicate code
- Alternative 2: Create a function to search for `Income` and `Expenditure` objects within the `IncomeListTotal` and `ExpLIstTotal` by date, respectively.
  - Pros: Allows for an Archive function to access all previous financial data by month
  - Cons: Difficult to implement, with commands requiring more processing overhead.

3.2. Archive Feature

This feature works concurrently with the money in/out feature to allow users to access all previous financial data by month.

It comprises of the following functions:

1. check income
2. check expenditure
3. list month income
4. list month expenditure

3.2.1. Check Income & Check Expenditure

The check commands are facilitated by the `ViewPastIncomeCommand` and `ViewPastExpenditureCommand` from the `Logic` Component.

Based on user input, `Logic` will either execute `ViewPastIncomeCommand` or `ViewPastExpenditureCommand.`
The command will search all `Income` or `Expenditure` objects in the corresponding List (`incomeListTotal` or `expListTotal`) in the `Model` component that is dated to the month specified in the user input.

Given below is an example usage scenario of how the check command behaves at each step, with variations depending on whether it is a `ViewPastIncomeCommand` or a `ViewPastExpenditureCommand`.

Step 1. The user calls either `ViewPastIncomeCommand` or `ViewPastExpenditureCommand` with the relevant input

- check income 10 2019 (`ViewPastIncomeCommand`)
- check expenditure 7 2018 (`ViewPastExpenditureCommand`)

Step 2. LogicManager calls MoneyParse command with user input.

Step 3. Paser is called and returns the corresponding list command (`ViewPastIncomeCommand`/`ViewPastExpenditureCommand`) object to LogicManager.

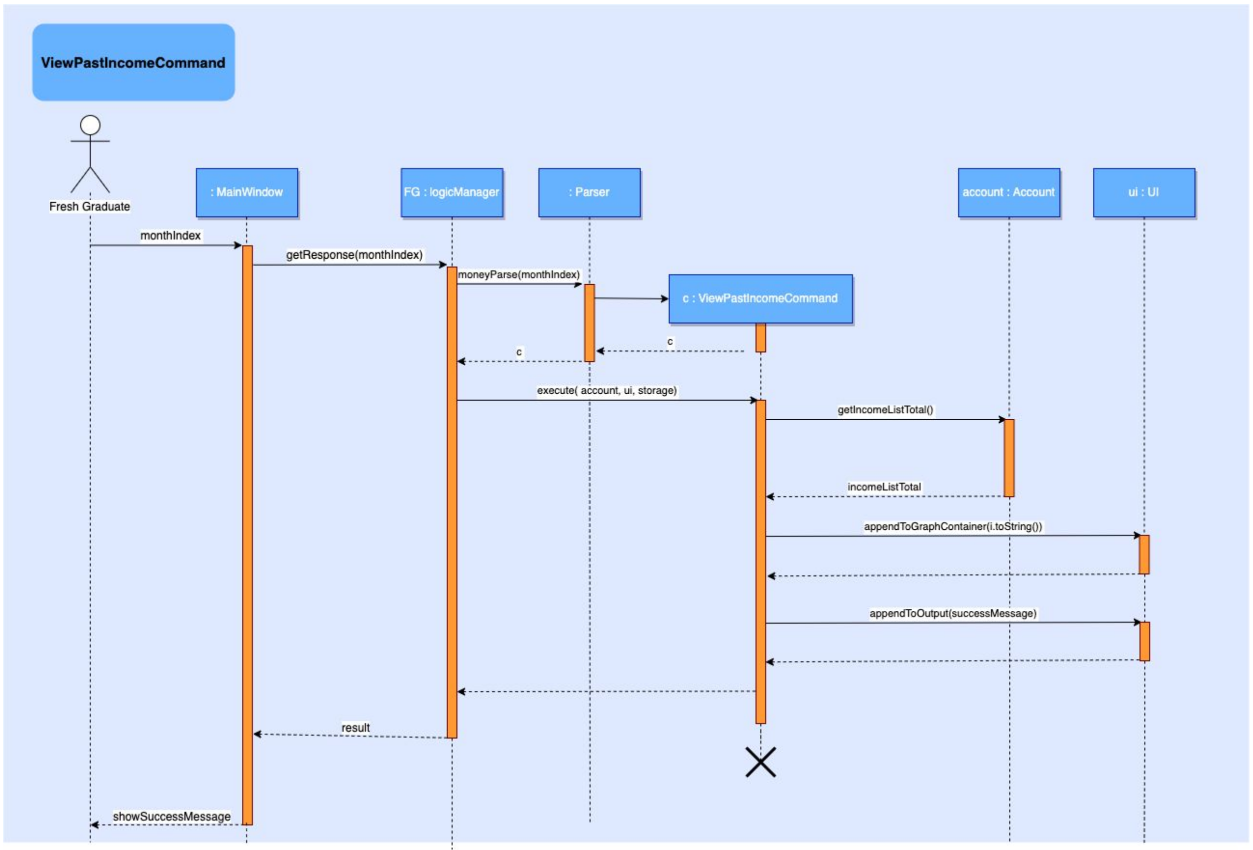
Step 4. The LogicManager calls method execute() on the object.

Step 5. Depending on the command called:

🗒	`ViewPastIncomeCommand`: Calls a method on the `Model` component to return `incomeListTotal` and searches for all `Income` objects with the attribute `payday` matching the month specified in the user input. `Calls appendToGraphContainer` on all `Income` objects found and calls `appendToOutput` a Success Message
🗒	`ViewPastExpenditureCommand`: Calls a method on the `Model` component to return `expListTotal` and searches for all `Expenditure` objects with the attribute `boughtDate` matching the month specified in the user input. Calls `appendToGraphContainer` on all `Expenditure` objects found and calls `appendToOutput` a Success Message

Step 6. The command will return the Success Message and the corresponding List which will be displayed in the GUI.

The following sequence diagram illustrates how the `DeleteExpenditureCommand` is carried out



NOTE: `ViewPastExpenditureCommand` follows the same sequence, albeit searching for `Expenditure` objects with the date specified in the user input from `ExpListTotal` with the method `getExpListTotal`

3.2.2. List Month Income & List Month Expenditure

The list month commands are likewise facilitated by the ViewPastIncomeCommand and ViewPastExpenditureCommand.

For the List Month Income function:

e.g. list month income

MoneyParser calls ViewPastIncomeCommand with the current month and year hard-coded as arguments to display the current month income to the user.

For the List Month Expenditure function:

e.g. list month expenditure

MoneyParser calls ViewPastExpenditureCommand with the current month and year hard-coded as arguments to display the current month expenditure to the user.

3.2.3. Design Considerations

- Alternative 1: List month income and list month expenditure was facilitated by another set of 2 ArrayLists containing `Income` objects and `Expenditure` objects.
  - Pros: Simpler implementation
  - Cons: Abundance of duplicated code, and required both lists to be repopulated for each list month command

- Alternative 2 (current choice): Hardcode the list month commands to the Archive command with the index of the current month
  - Pros: Reduces number of commands and duplicated implementation
  - Cons: Hard to implement

3.3. Goal Setting Feature

This feature allows users to manage and set their financial goals.

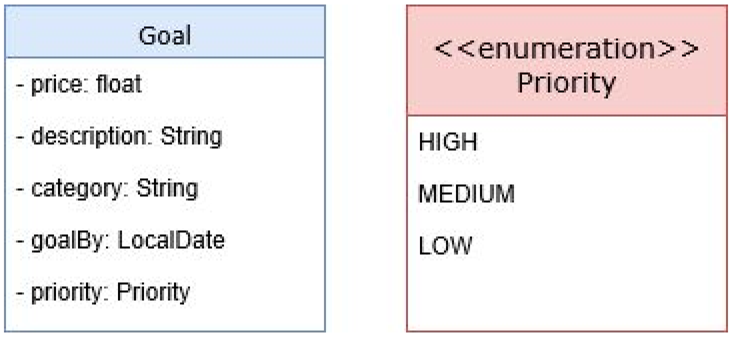
It comprises of the following functions:

1. Add Goal
2. Complete Goal
3. Commit Goal
4. Delete Goal
5. List Goal

3.3.1. Add Goal Function

The add goal function is facilitated by the AddGoalCommand from the Logic Component. A Goal object is instantiated and this object has price, description, category, target date and priority level as its attributes. This object is then added into Accounts under GoalList.

The following diagram illustrates the Goal objects:



Given below is an example usage scenario of how the add goal function behaves at each step.

Step 1: The user calls the add goal command from the GUI with its respective parameters e.g goal Motorbike /amt 10000 /by 14/3/2022 /priority HIGH.

Step 2: The LogicManager calls moneyParse on Parser with the user input.

Step 3: Parser is called and it constructs a AddGoalCommand object. The AddGoalCommand Object is returned to LogicManager.

Step 4: The LogicManager calls execute() on the AddGoalCommand object.

Step 5: The Logic Component will interact with Account to add the goal.

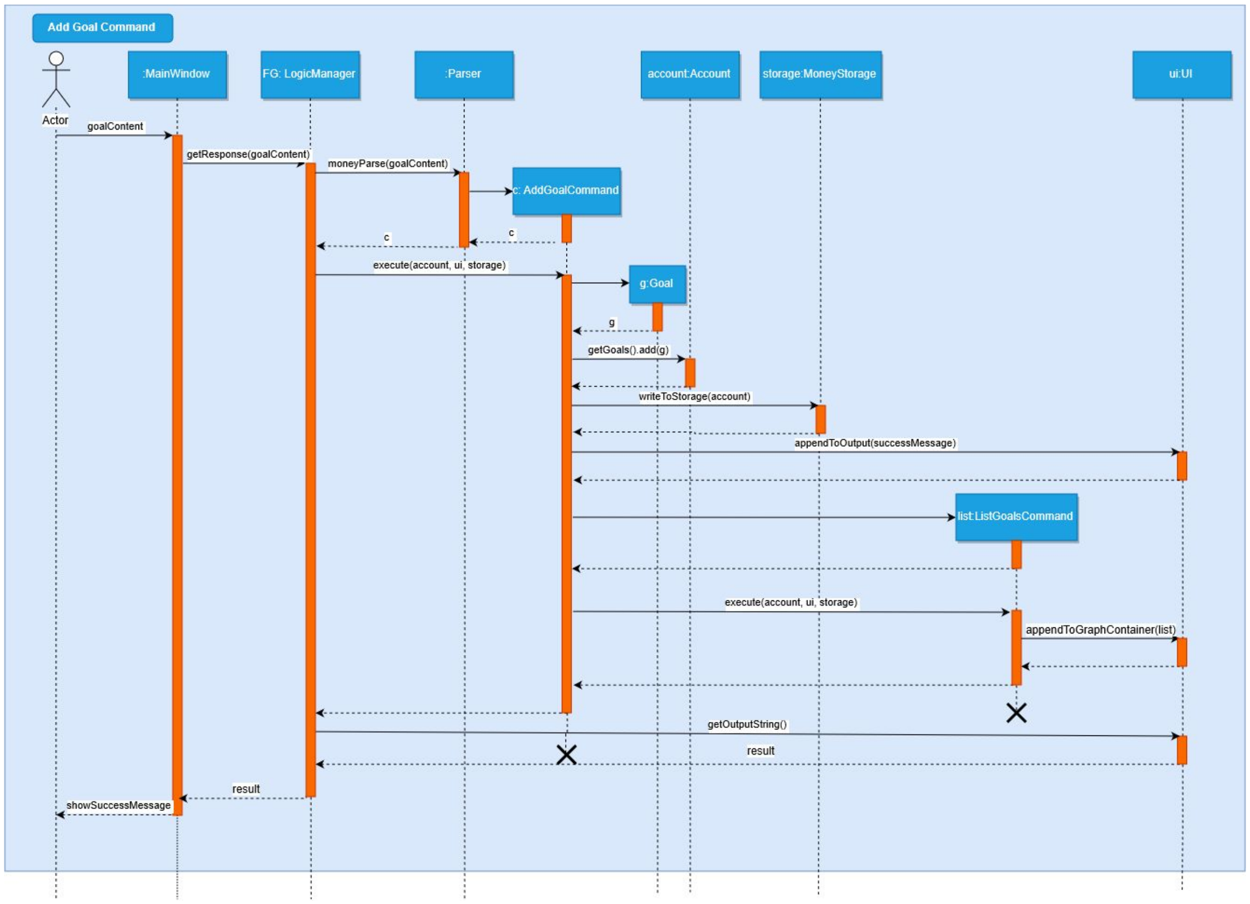
Step 6: AddGoalCommand will call appendToOutput a Success Message.

Step 7: AddGoalCommand object then constructs a ListGoalsCommand object and calls execute on it.

Step 8: ListGoalsCommand will call appendToOutput a Success Message and call appendToGraphContainer the updated Goal List.

Step 9: The command result will return the success message and the updated Goal list which will be displayed in the GUI.

The following sequence diagram illustrates how the Add Goal function works:



3.3.2. Done Goal Function

The Done Goal Function is facilitated by the DoneGoalCommand from the Logic Component (FG). Upon executing the DoneGoalCommand, the Goal set by the user will be converted into an expenditure object and added into the expenditure list. Then, the Goal will be subsequently removed from the MoneyAccount.txt in the data folder.

Given below is an example usage scenario and how the Done Goal Function behaves at each step.

Step 1: The user calls the DoneGoalCommand with the Goal's index in the Goal List.

Step 2: The Logic manager (FG) calls moneyParse on Parser with the user input.

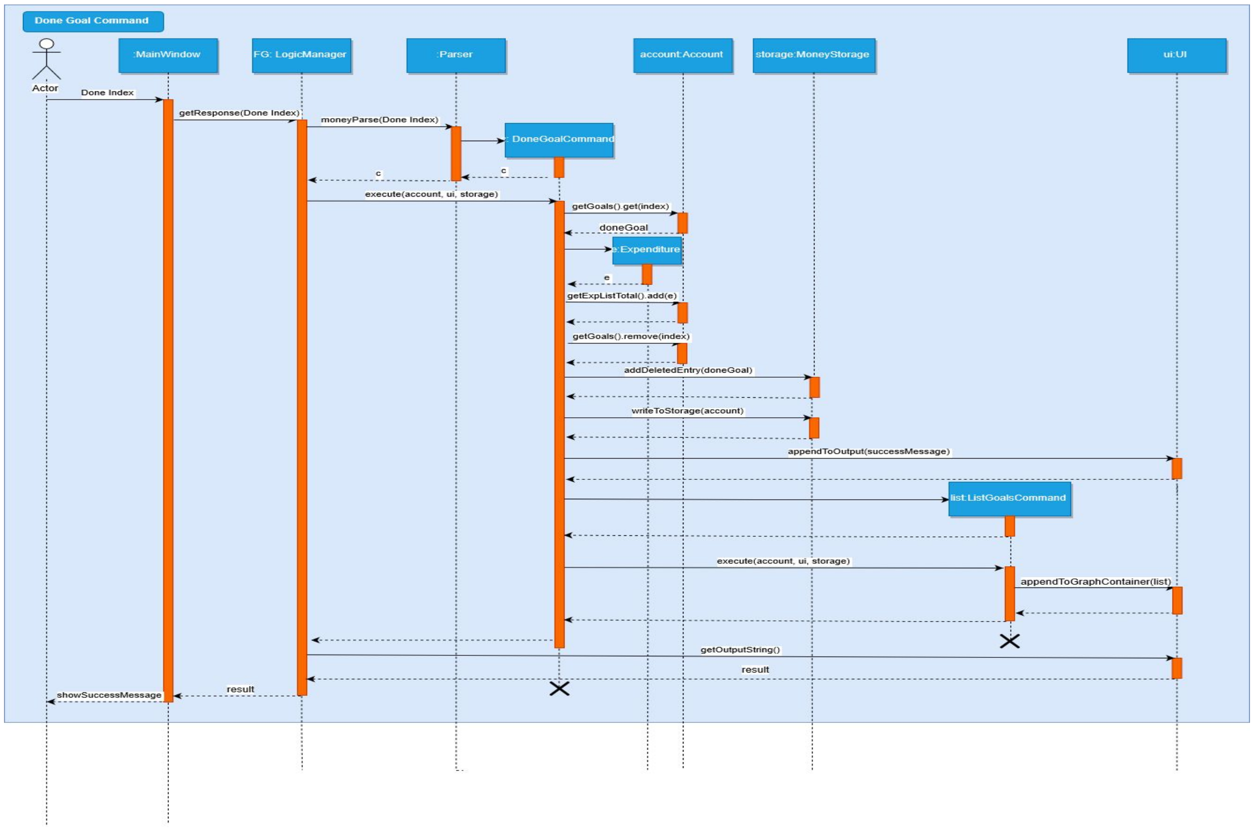
Step 3: Parser is called and constructs a DoneGoalCommand object and returns it to Logic Manager(FG).

Step 4: The Logic Manager will call execute() on the DoneGoalCommand object. If the corresponding index is invalid, it will return an error message of "ERROR: The serial number of the Goal is Out Of Bounds!". Furthermore, if the user cannot afford the Goal to be completed, it will also return an error message of “ERROR: Goal Price exceeds Goal Savings”.

Step 5: The Logic Component then interacts with the Account model to convert the Goal into an expenditure object and add it into the expenditure list. Then, it will remove the goal from the goal list.

Step 6: The Command result would then return the success message then the updated goal list will be displayed in the GUI.

The following sequence diagram illustrates how the DoneGoal function works:



3.3.3. Commit Goal Function

The Commit Goal Function is facilitated by the CommitGoalCommand from the Logic component. It allows the user to select multiple Goals he/she would like to complete and will display the changes to the user’s finances if they were to complete the selected goals.

Given below is an example usage scenario and how the CommitGoalCommand behaves at each step.

Step 1: The user calls the CommitGoalCommand with the index of the Goals that they would like to commit. E.g commit goal 2,3,4.

Step 2: The Logic manager (FG) calls moneyParse on Parser with the user input.

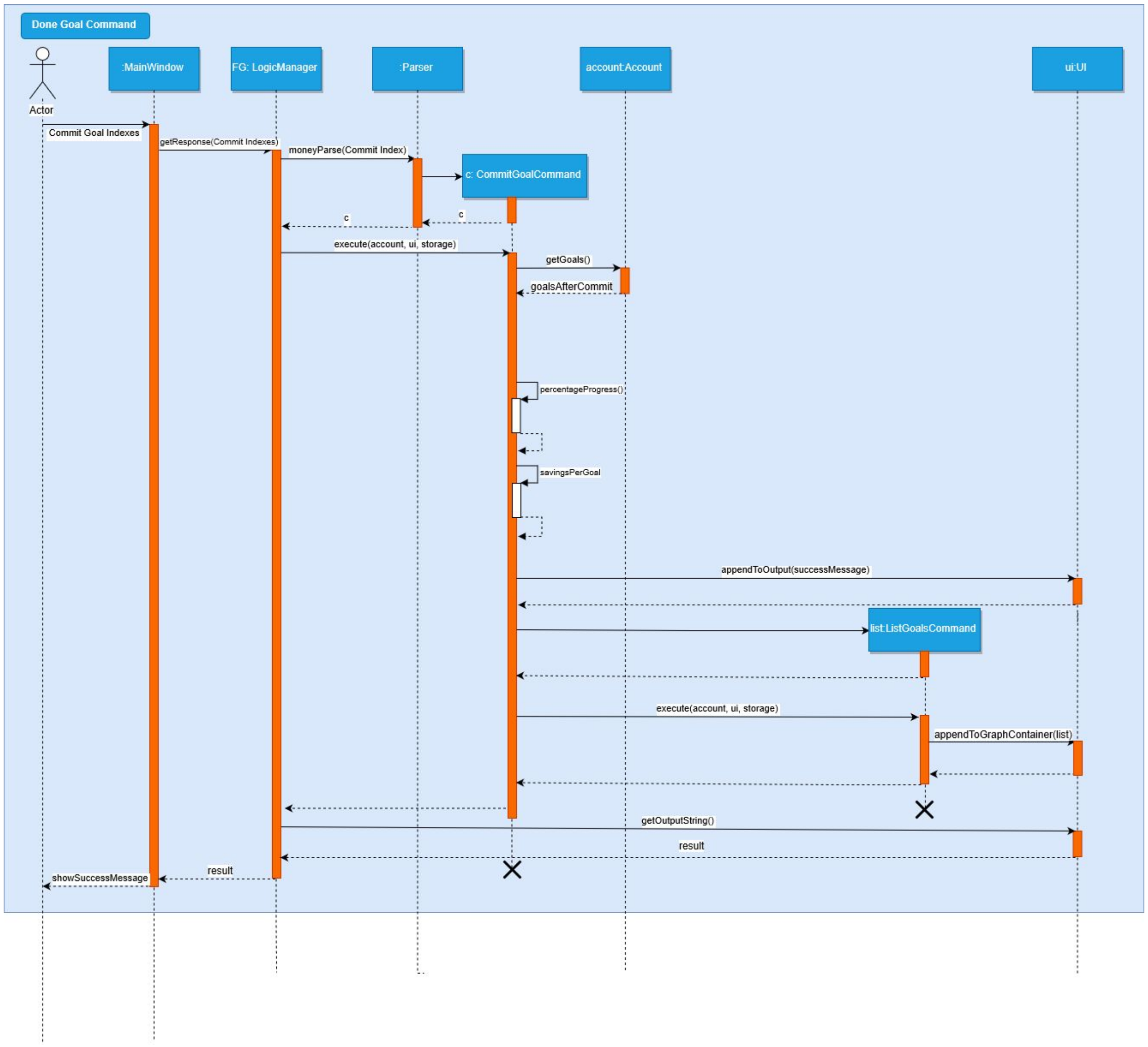
Step 3: Parser is called and constructs a CommitGoalCommand object and returns it to Logic Manager(FG).

Step 4: The Logic Manager will call execute() on the CommitGoalCommand object. If the corresponding index is invalid, it will return an error message of “ERROR: The serial number of the Goal is Out Of Bounds!”. Furthermore, if the user cannot afford the Goal to be completed, it will also return an error message of “ERROR: Goal Price exceeds Goal Savings”. In addition, if the user did not key in any goals to commit, it will return an error message of “ERROR: The Description of the command cannot be empty”.

Step 5: CommitGoalCommand will compute the simulated finances of the user and the items left in the Goal List after the commit.

Step 6: The command result will be returned a success message as well as the simulated financial numbers and the Goal List after a commit.

The following sequence diagram illustrates how the Commit Goal function works:



3.3.4. Delete Goal Function

The Delete Goal Function is facilitated by the DeleteGoalCommand from the Logic component. It allows the user to select Goals he/she would like to delete and will display the updated goal list upon deletion.

Given below is an example usage scenario and how the Delete Goal Function behaves at each step.

Step 1: The user calls the DeleteGoalCommand with the Goal’s index in the Goal List.

Step 2: The Logic manager (FG) calls moneyParse on Parser with the user input.

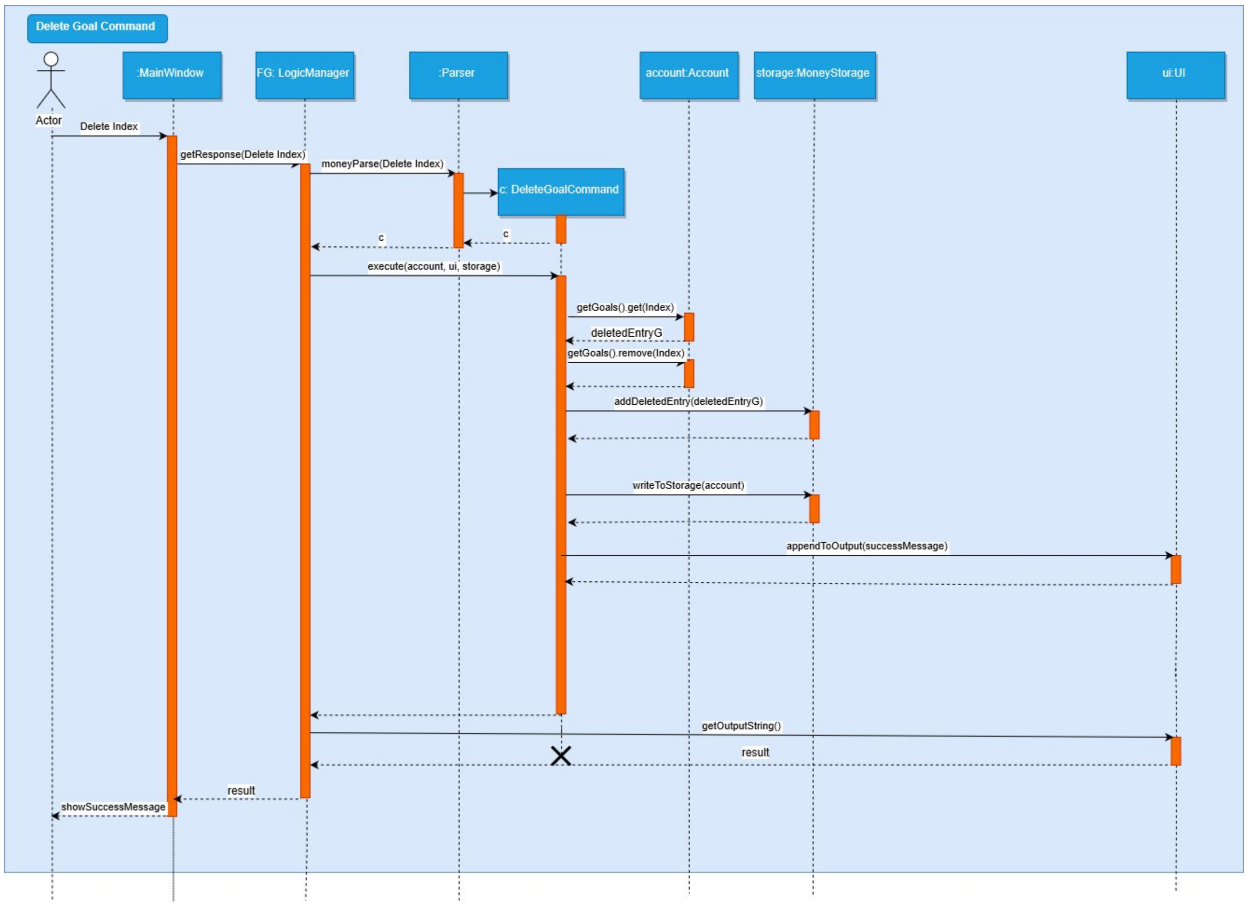
Step 3: Parser is called and constructs a DeleteGoalCommand object and returns it to Logic Manager(FG).

Step 4: The Logic Manager will call execute() on the DeleteGoalCommand object. If the corresponding index is invalid, it will return an error message of “ERROR: The serial number of the Goal is Out Of Bounds!”.

Step 5: The Logic Component then interacts with the Account model to remove the goal from the goal list.

Step 6: The Command result would then return the success message then the updated goal list will be displayed in the GUI.

The following sequence diagram illustrates how the Commit Goal function works:



3.3.5. Add Goal Function

The List Goal Function is facilitated by the ListGoalCommand from the Logic component. It allows the user to list the goals that he/she has set for themselves.

Given below is an example usage scenario of how the list goal function behaves at each step.

Step 1: The user calls the list goal command from the GUI with its respective parameters e.g list goals

Step 2: The LogicManager calls moneyParse on Parser with the user input.

Step 3: Parser is called and it constructs a ListGoalCommand object. The ListGoalCommand Object is returned to LogicManager.

Step 4: The LogicManager calls execute() on the ListGoalCommand object.

Step 5: The Logic Component will interact with Account to retrieve the goal list.

Step 6: ListGoalsCommand will call appendToOutput a Success Message and call appendToGraphContainer the updated Goal List.

Step 7: The command result will return the success message and the Goal list which will be displayed in the GUI.

3.3.6. Design Considerations

Aspect: Displaying an updated list to the user every time the goal list is updated.

- Alternative 1 (current choice): Call the list command every time the list is updated.
    - Pros: Easy to implement.
    - Cons: Increased coupling between commands.
- Alternative 2: The list will persist and only updates when it detects a change in its contents.
    - Pros: Little to no coupling between components.
    - Cons: ensure that it is able to detect all types of changes to the list e.g add, delete, done etc.

3.4. Loan Feature

This feature allows users to track their incoming/outgoing loans.
The Loan feature is facilitated by the `AddLoanCommand, SettleLoanCommand, ListLoanCommand and DeleteLoanCommand` from the `Logic` component.

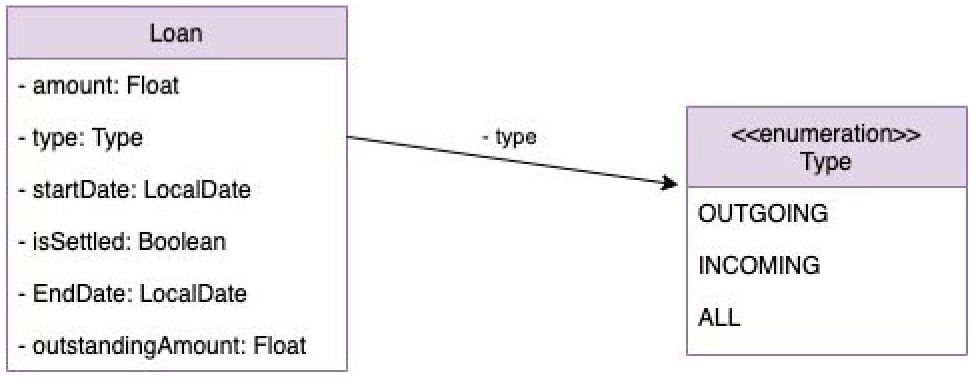
It comprises of the following functions:

1. Add incoming/outgoing loan
2. List all/incoming/outgoing loans
3. Settle incoming/outgoing loan
4. Delete incoming/outgoing loan

3.4.1 Add Incoming/Outgoing Loan Function

The add command is facilitated by the `AddLoanCommand` from the Logic component. A Loan object is instantiated to add an incoming/outgoing loan into an ArrayList of Loans in the Model component.

The following Diagram describes the Loan class:



Given below is an example usage scenario and how the loan feature behaves at each step.

Step 1. The user calls the AddLoanCommand with the relevant parameters to create a loan.
E.g. borrowed parents /amt 5000 /on 9/10/2019 (for incoming loan)
E.g. lent friend-A /amt 400 /on 10/10/2019 (for outgoing loan)
Note: isSettled, endDate, outstandingAmount parameters are not included as these attributes are only updated as the loan is settled

Step 2. The LogicManager calls MoneyParse command with the user input.

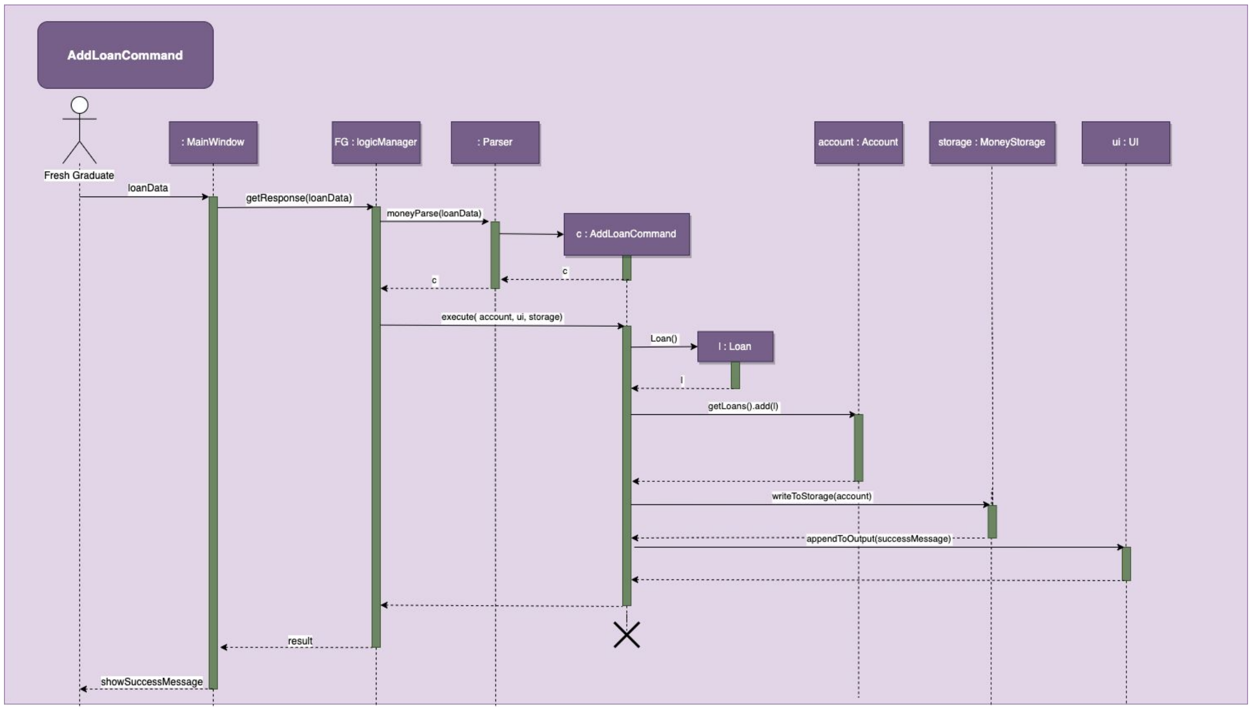
Step 3. Parser is called and returns a AddLoanCommand object to LogicManager.

Step 4. The LogicManager calls method execute() on AddLoanCommand object.

Step 5. The AddLoanCommand object creates a Loan object which is added into the ArrayList of Loans in Model

Step 6. AddLoanCommand calls appendToOutput a Success Message which is displayed in the GUI.

The following sequence diagram illustrates the AddLoanCommand



3.4.2 Settle Incoming/Outgoing Loan Function

The add command is facilitated by the `SettleLoanCommand` from the Logic component. An existing Loan object is retrieved from the ArrayList of Loans in the Model component and the debt of the loan is settled according to the amount inputted by the user.

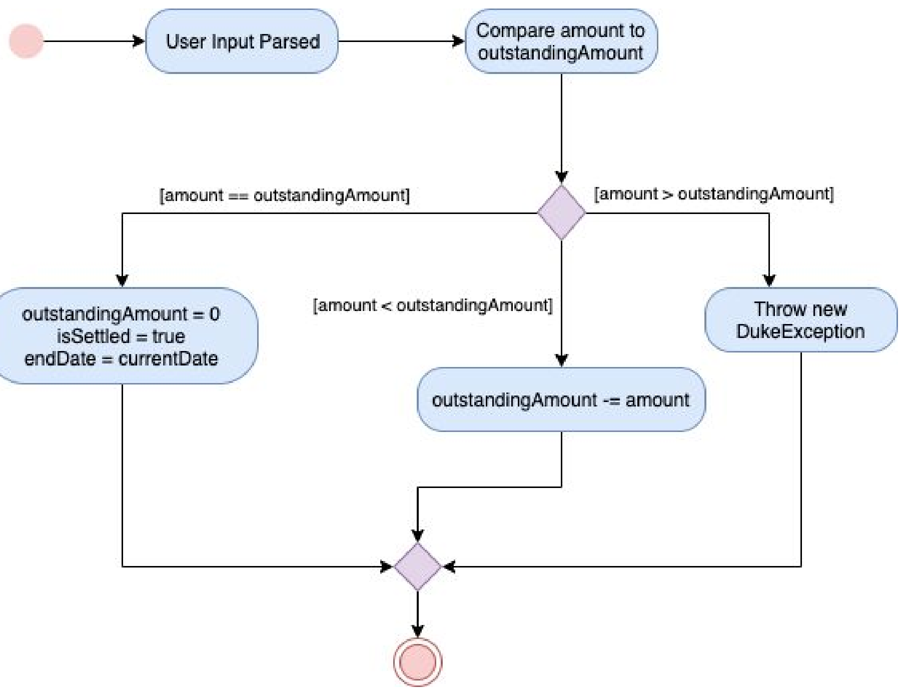
Step 1. The user calls SettleLoanCommand with the relevant parameters to settle a loan.
E.g. paid 400 /to parents (for incoming loan)
E.g received 300 /from friend-A (for outgoing loan)
Note: User can choose to input the name or index of the loan party in the Incoming/Outgoing Loan List

Step 2. LogicManager calls MoneyParse command with user input.

Step 3. Paser is called and returns a SettleLoanCommand object to LogicManager.

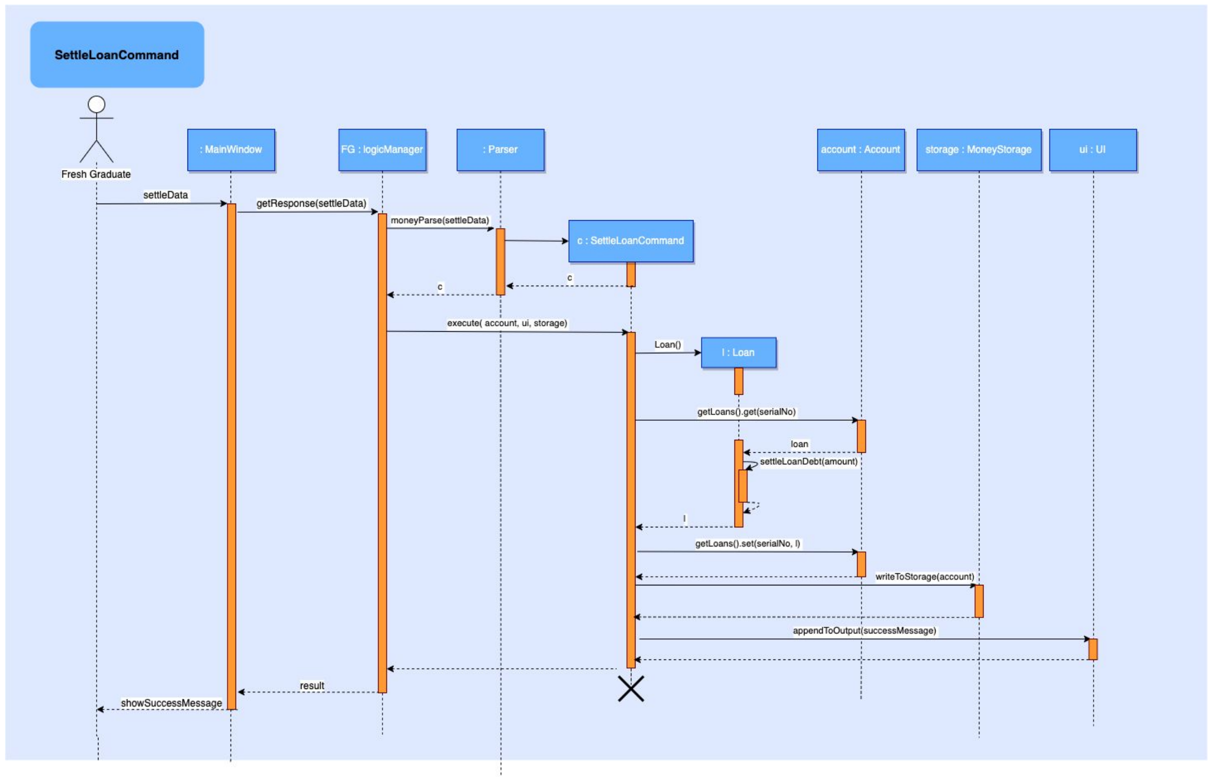
Step 4. The LogicManager calls method execute() on SettleLoanCommand object.

Step 5. The SettleLoanCommand object interacts with Model and searches for the Loan object specified in the user input in the ArrayList of Loans. The `settleLoanDebt` method is called on the Loan object found with the amount in user input used as an argument to settle the debt.
The diagram below shows the behavior of the `settleLoanDebt` method


User Input Parsed
Compare amount to outstandingAmount
[amount == outstandingAmount]
[amount > outstandingAmount]
[amount < outstandingAmount]
outstandingAmount = 0
isSettled = true
endDate = currentDate
outstandingAmount -= amount
Throw new DukeException

Step 6. The SettleLoanCommand object interacts with Model and adds Income/Expenditure Object depending on user input querying an outgoing/incoming loan and adds to the ArrayList of Income/Expenditure in the Model component, respectively.

Step 7. The SettleLoanCommand object will call writeToFIle to store any changes in Storage and appendToOutput a Success Message which is displayed in the GUI.



3.4.3 List Incoming/Outgoing Loans Function

The list command is facilitated by the `ListLoansCommand` from the Logic component. User chooses between 3 possible commands to list outgoing/incoming/all Loan Objects in the ArrayList of Loans in the Model component.

Step 1. The user calls ListLoanCommand with the relevant input to list loans in Model according to type
E.g. list all loans (to list both types of loans)
E.g. list incoming loans (to list only incoming loans)
E.g. list outgoing loans (to list only outgoing loans)

Step 2. LogicManager calls MoneyParse command with user input.

Step 3. Paser is called and returns a ListLoanCommand object to LogicManager.

Step 4. The LogicManager calls method execute() on ListLoanCommand object.

Step 5. ListLoanCommand will call appendToOutput a Success Message and call appendToGraphContainer the All/Incoming/Outgoing Loan List.

Step 6. The command will return the success message and the corresponding Loan list which will be displayed in the GUI.

3.4.4 Delete Loan Function

The delete loan command is facilitated by the `DeleteLoanCommand` from the `Logic` Component.

Capable of deleting both incoming and outgoing loans, a `Loan` object within the `loans` (ArrayList of Loan) in the `Model` component is removed according to the serial number provided in the user input.

The serial number used by the `DeleteLoanCommand` is the serial number of the loans within the loan list, and not the incoming loan list or the outgoing loan list.

Step 1: The user calls the delete command from the GUI with the serial number of the object within the LoanList to be deleted.
e.g. delete loan 2

Step 2: The LogicManager calls moneyParse on Parser with the user input.

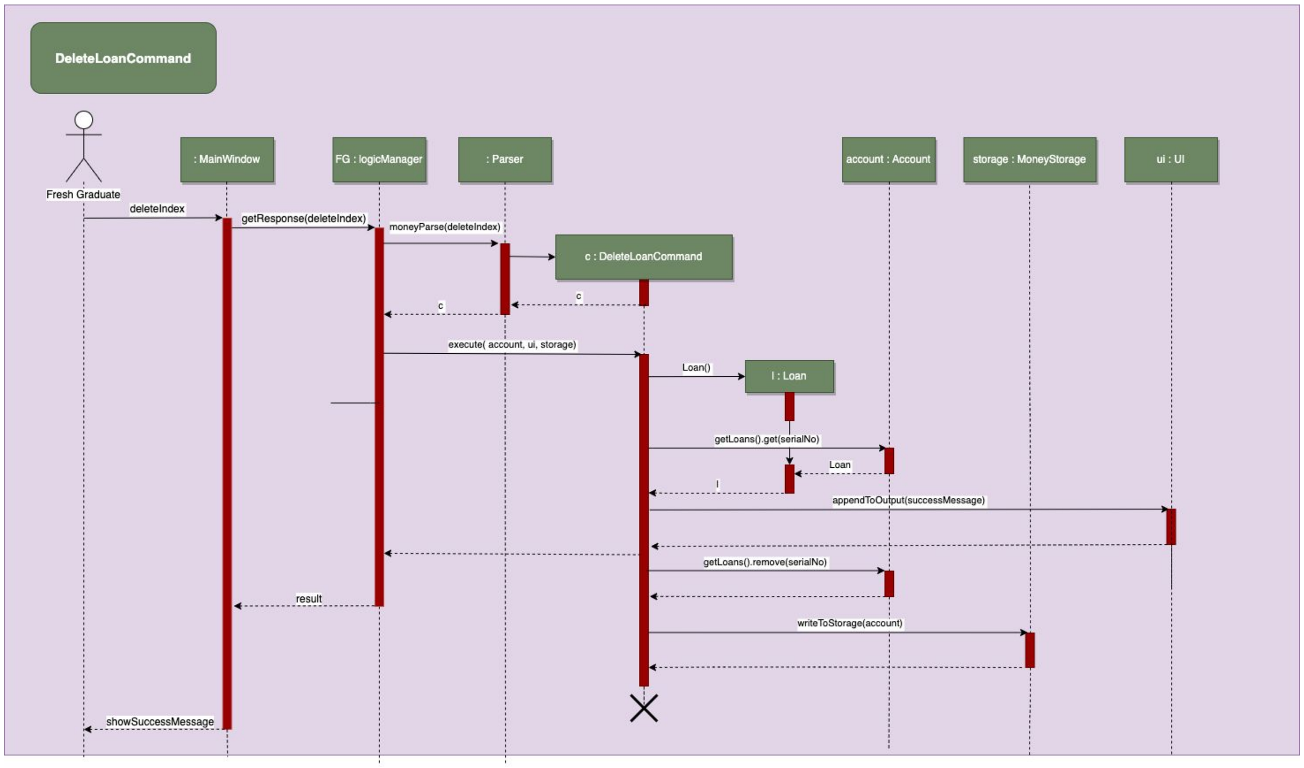
Step 3: Parser is called and it constructs either a `DeleteLoanCommand` object, which is returned to LogicManager.

Step 4: The LogicManager calls execute() on the `DeleteLoanCommand` object.

Step 5: The `Logic` Component will interact with the `Model` component to return `loans` and calls `remove(serialNo)` on the `Loan` Object to remove it from `loans`.

Step 6: `DeleteLoanCommand` will call appendToOutput a Success Message.

The following sequence diagram illustrates how the `DeleteLoanCommand` is carried out



3.4.5 Design Considerations

During the development of the Loans feature, multiple designs were considered before settling on the current implementation.

- Alternative 1: Creating separate classes for incoming and outgoing loans that inherit the Loan class.
  - Pros: Allow incoming loans and outgoing loans to take on unique attributes
  - Cons: Each type of loan will require its own set of classes for the add, settle and list commands
- Alternative 2 (current choice): Combine both incoming and outgoing loans into a single class Loan differentiated by an Enumeration
  - Pros: Reduces the number of command classes and duplicate code
  - Cons: Difficult to implement, with the code for add, settle and list commands becoming more complex

3.5. Instalment Feature

This feature allows users to manage and set their instalments.

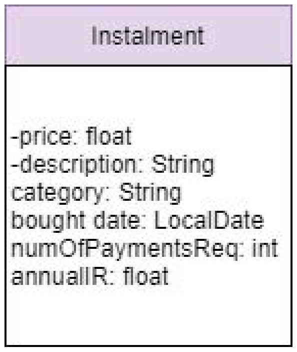
It comprises of the following functions:

1. **Add Instalment Function**
2. **Automatically Update Instalment payments Feature**
3. **Delete Instalment Function**
4. **List Instalments Function**

3.5.1. Add Instalment Function

The add instalment function is facilitated by the AddInstalmentCommand from the Logic Component. An Instalment object is instantiated and this object has price, description, category, item bought date, number of payments in total and annual interest rate as its attributes. This object is then added into Accounts under InstalmentList.

The following diagrams describes the Instalment Class.



Given below is an example usage scenario of how the add instalment function behaves at each step.

Step 1: The user calls the add instalment command from the GUI with its respective parameters.

E.g `add instalment mortgage /amt 100000 /within 200 months /from 12/12/2010 /percentage 6`

Step 2: The LogicManager calls moneyParse on Parser with the user input.

Step 3: Parser is called and it constructs a AddInstalmentCommand object. The AddInstalmentCommand Object is returned to LogicManager.

Step 4: The LogicManager calls execute() on the AddInstalmentCommand object.

Step 5: The Logic Component will interact with Account to add the Instalment.

Step 6: The Storage component will overwrite the original txt file with the new list which contains the added instalment.

Step 7: AddInstalmentCommand will call appendToOutput a Success Message.

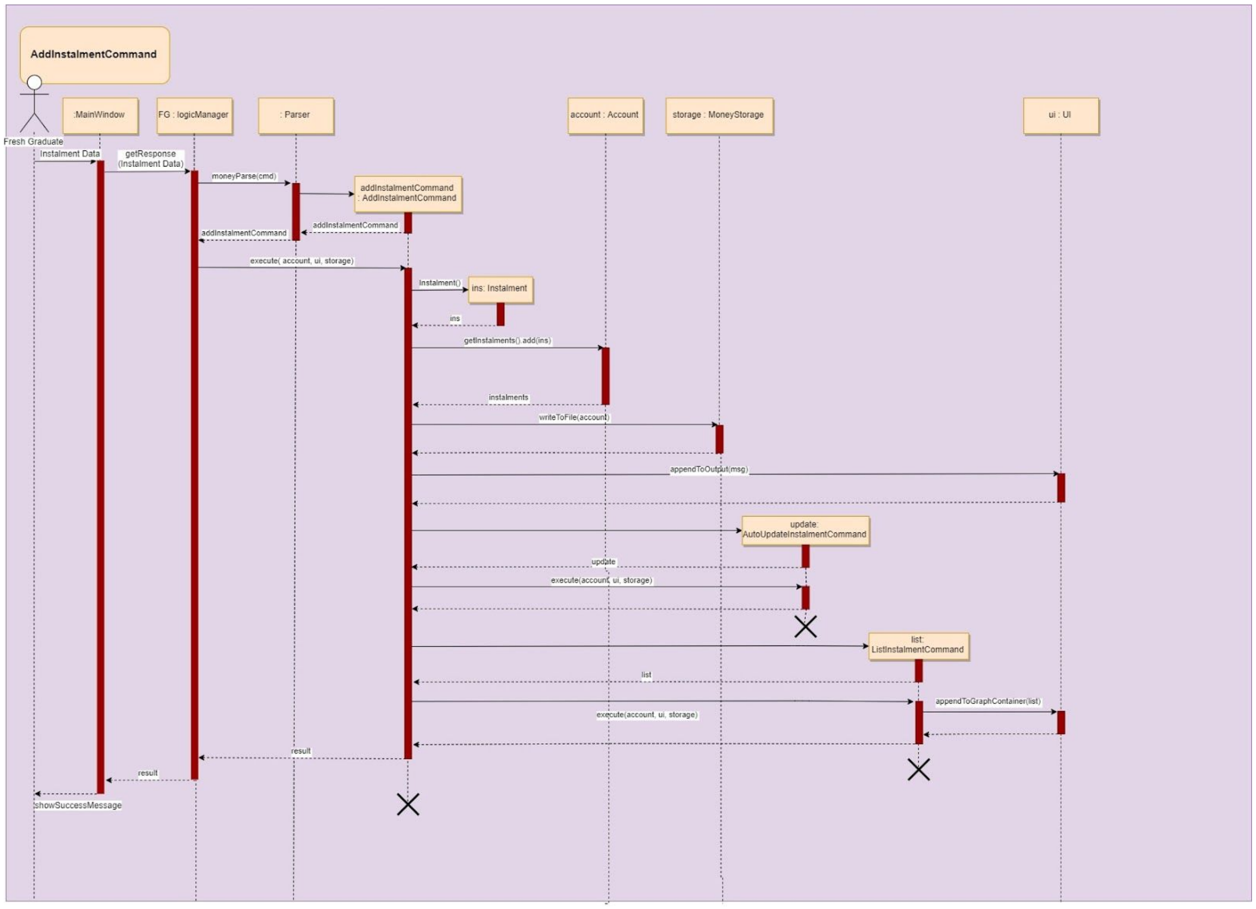
Step 8: AddInstalmentCommand object then constructs a AutoUpdateInstalmentCommand object and calls execute on it.

Step 9: AddInstalmentCommand object then constructs a ListInstalmentsCommand object and calls execute on it.

Step 10: ListInstalmentsCommand will call appendToOutput a Success Message and call appendToGraphContainer the updated Instalment List.

Step 11: The command result will return the success message and the updated Instalment list which will be displayed in the GUI.

Below shows the sequence diagram of AddInstalmentCommand.



3.5.2. Automatically Update Instalment payments Feature

The auto update instalment payments feature is facilitated by the AutoUpdateInstalmentCommand from the Logic Component. An Instalment object is already instantiated. An AutoUpdateInstalmentCommand object has current date that it is created as it’s attributes. All Instalment objects is automatically pushed as an expenditure on it’s payment date every month.

Given below is an example usage scenario of how the auto update instalment payments feature behaves at each step.

Step 1: The user calls the any command from the GUI with its respective parameters.

Step 2: The LogicManager automatically instantiates an AutoUpdateInstalmentCommand object and calls executes() on the AutoUpdateInstalmentObject.

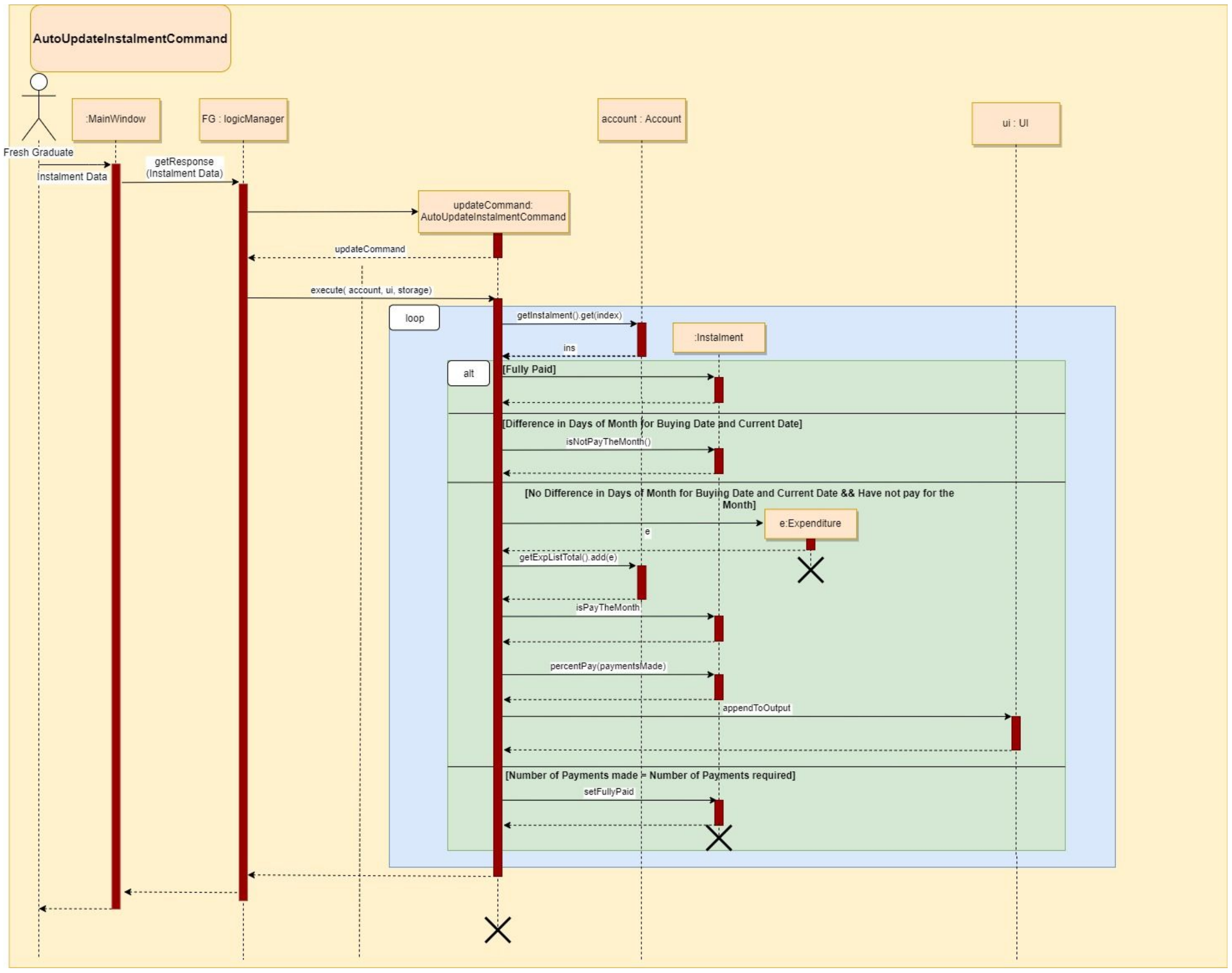
Step 3: The Logic Component will interact with account and extract all Instalment objects and iterate through the list of instalments.

Step 4: If the current date is equal to the pay date of a particular instalment object, and the payment is not done, the Logic Component will interact with Account to add the instalment as an expenditure.

Step 5: The Storage component will overwrite the original txt file with the new list which contains the added expenditure and updated instalment payment.

Step 6: Step 4 to 5 is repeated for all the instalment object till all of them is checked through.

Below shows the sequence diagram of AutoUpdateInstalmentCommand.


AutoUpdateInstalmentCommand
Fresh Graduate
:MainWindow
FG : logicManager
account : Account
ui : UI
Instalment Data
getResponse (Instalment Data)
updateCommand: AutoUpdateInstalmentCommand
updateCommand
execute(account, ui, storage)
loop
getInstalment().get(index)
:Instalment
ins
alt
[Fully Paid]
[Difference in Days of Month for Buying Date and Current Date]
isNotPayTheMonth()
[No Difference in Days of Month for Buying Date and Current Date && Have not pay for the Month]
e:Expenditure
e
getExpListTotal().add(e)
isPayTheMonth
percentPay(paymentsMade)
appendToOutput
[Number of Payments made = Number of Payments required]
setFullyPaid

3.5.3. Delete Instalment Function

The Delete Instalment Function is facilitated by the DeleteInstalmentCommand from the Logic component. It allows the user to select Instalment he/she would like to delete and will display the updated Instalment list upon deletion.

Given below is an example usage scenario and how the Delete Instalment Function behaves at each step.

Step 1: The user calls the DeleteInstalmentCommand with the Instalment’s index in the Instalment List.

Eg. `delete instalment 3`

Step 2: The Logic manager (FG) calls moneyParse on Parser with the user input.

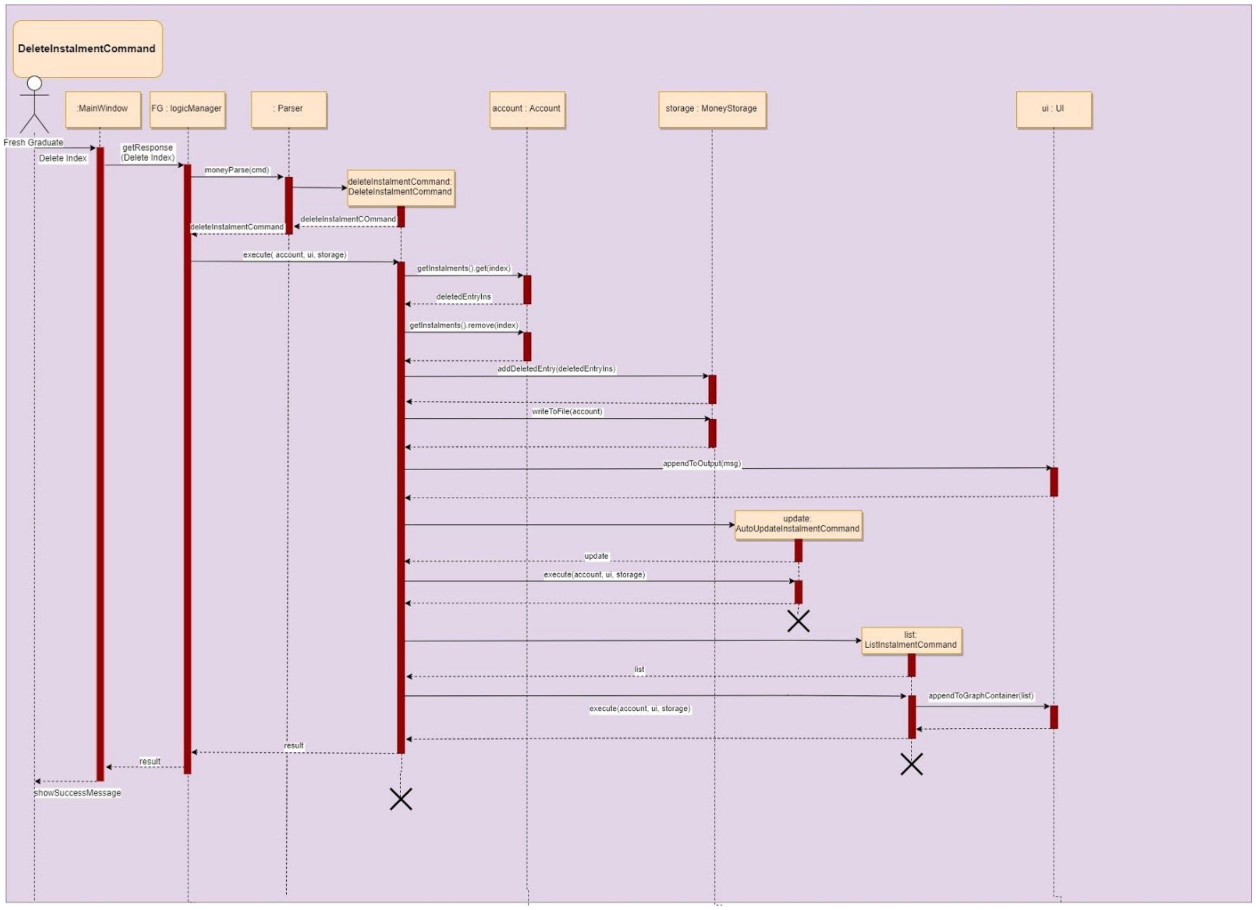
Step 3: Parser is called and constructs a DeleteInstalmentCommand object and returns it to Logic Manager(FG).

Step 4: The Logic Manager will call execute() on the DeleteInstalmentCommand object. If the corresponding index is invalid, it will return an error message of “ERROR: The serial number of the Instalment is Out Of Bounds!”.

Step 5: Else, the Logic Component then interacts with the Account model to remove the particular instalment from the Instalments list.

Step 6: The Command result would then return the success message, the updated instalments list will then be displayed in the GUI.

Below shows the sequence diagram of DeleteInstalmentCommand.


DeleteInstalmentCommand
Fresh Graduate
:MainWindow
FG : logicManager
: Parser
account : Account
storage : MoneyStorage
ui : UI
Delete Index
getResponse (Delete Index)
moneyParse(cmd)
deleteInstalmentCommand: DeleteInstalmentCommand
deleteInstalmentCOmmand
deleteInstalmentCommand
execute(account, ui, storage)
getInstalments().get(index)
deletedEntryIns
getInstalments().remove(index)
addDeletedEntry(deletedEntryIns)
writeToFile(account)
appendToOutput(msg)
update: AutoUpdateInstalmentCommand
update
execute(account, ui, storage)
list: ListInstalmentCommand
list
appendToGraphContainer(list)
execute(account, ui, storage)
result
result
showSuccessMessage

3.5.4. List Instalment Function

The List Instalment Function is facilitated by the ListInstalmentCommand from the Logic component. It allows the user to list the instalments that he/she is required to pay.

Given below is an example usage scenario of how the list instalment function behaves at each step.

Step 1: The user calls the list instalment command from the GUI with its respective parameters e.g list all instalments

Step 2: The LogicManager calls moneyParse on Parser with the user input.

Step 3: Parser is called and it constructs a ListInstalmentCommand object. The ListInstalmentCommand Object is returned to LogicManager.

Step 4: The LogicManager calls execute() on the ListInstalmentCommand object.

Step 5: The Logic Component will interact with Account to retrieve the Instalment list.

Step 6: ListInstalmentsCommand will call appendToOutput a Success Message and call appendToGraphContainer the updated Instalment List.

Step 7: The command result will return the success message and the Instalment list which will be displayed in the GUI.

3.5.5. Design Considerations

- **Implementation of AddInstalmentCommand**

Alternative 1: Overriding execute() (Current Choice)

Pros: Easy for developers to understand as all commands override execute().

Cons: All classes of commands that is part of the moneycommands package must write their own overriding method of execute().

Alternative 2: Writing individual executeAddInstalmentCommand()

Pros: Can cater to the needs of adding instalments directly and do not have to implement it in every moneycommands.

Cons: Harder for developers to understand.

- **Saving of new Instalment to txt file**

Alternative 1: Override entire text file (Current Choice)

Pros: Easy for developers to implement.

Cons: May have performance issues in terms of memory usage.

Alternative 2: Append to text file.

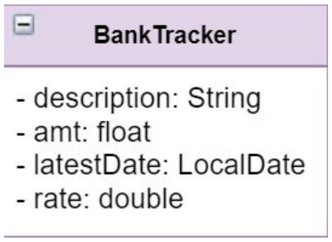
Pros: Efficient data storage technique.

Cons: Implementation issues and is hard to track the respective data files of various types.

3.6. Bank Account Feature

This feature helps to track the information about user’s bank accounts. All the information of one bank account will be stored in one `BankTracker` which includes 4 types of data (Figure ?):

1. The description of this account
2. The current balance
3. The latest date when update this account
4. The interest rate for this account



Based on the `BankTracker`, there are 5 commands to manipulate the bank account trackers:

1. `CreateBankAccountCommand`: This command helps to create a new bank account tracker.
2. `DeleteBankAccountCommand`: This command deletes the bank account tracker according to the given index number of the tracker.
3. `ListBankTrackerCommand`: This command lists down all the existing bank account trackers for users to check.
4. `CheckFutureBalanceCommand`: This command will show users the balance of an account at a specific future date with current balance and interest rate.
5. `InternalTransferCommand`: This command simulates withdraw/deposit operations on the bank accounts.

3.6.1. Create Bank Account Trackers

This feature allow the user to track the bank account information.

Step1. MainWindow class will handle this command by invoking getResponse method from logic component to get a string of message.

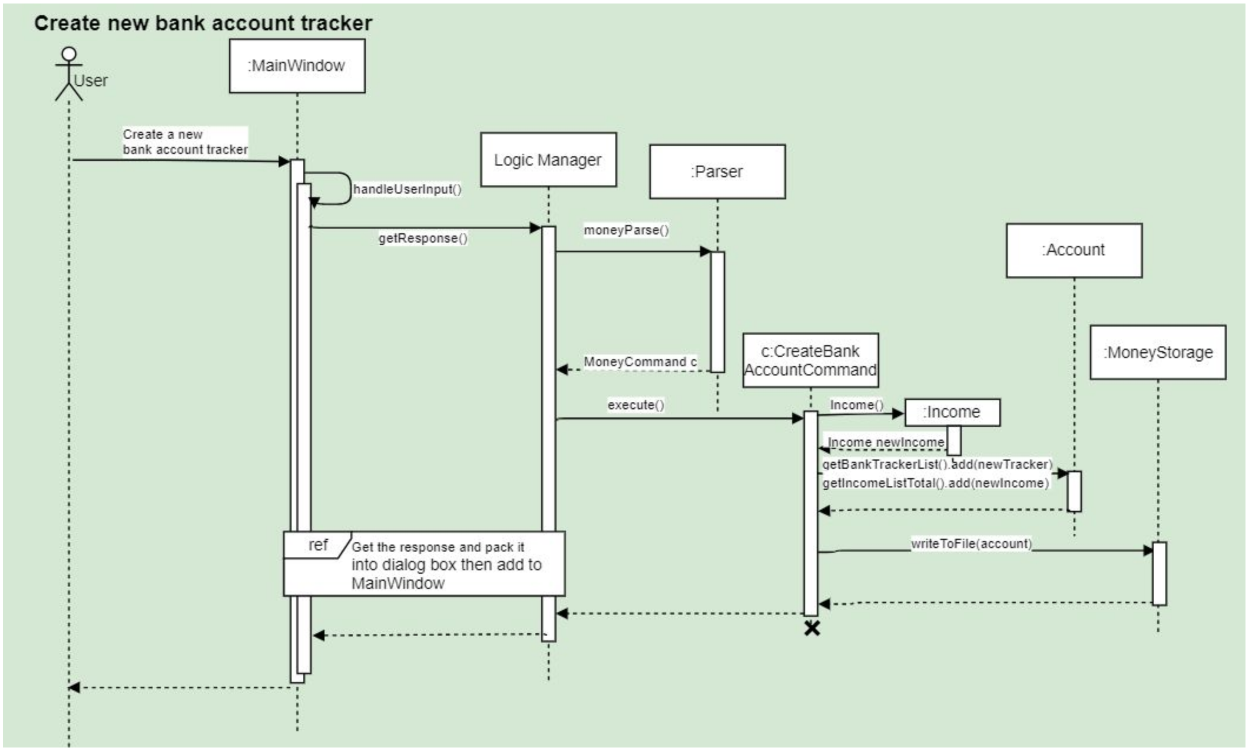
Step2. The input will be parsed and get a CreateBankAccountCommand, then execute method in CreateBankAccountCommand will be invoked.

Step3. During invoking the execute method, a new bank tracker will be created and added to the bank tracker list.

Step4. Then a new income will be created because of the initial balance of the tracker and added into the overall list.

Step5. Finally, MoneyStorage will store the changes and the details of the new tracker will be printed.

The graph below demonstrates the CreateBankAccountCommand.


Create new bank account tracker
User
:MainWindow
Logic Manager
:Parser
:Account
c:CreateBank AccountCommand
:MoneyStorage
:Income
Create a new bank account tracker
handleUserInput()
getResponse()
moneyParse()
MoneyCommand c
execute()
Income()
Income newIncome
getBankTrackerList().add(newTracker)
getIncomeListTotal().add(newIncome)
writeToFile(account)
ref
Get the response and pack it into dialog box then add to MainWindow

3.6.2. Delete Bank Account Tracker

This feature allow user to delete the existing bank account tracker.

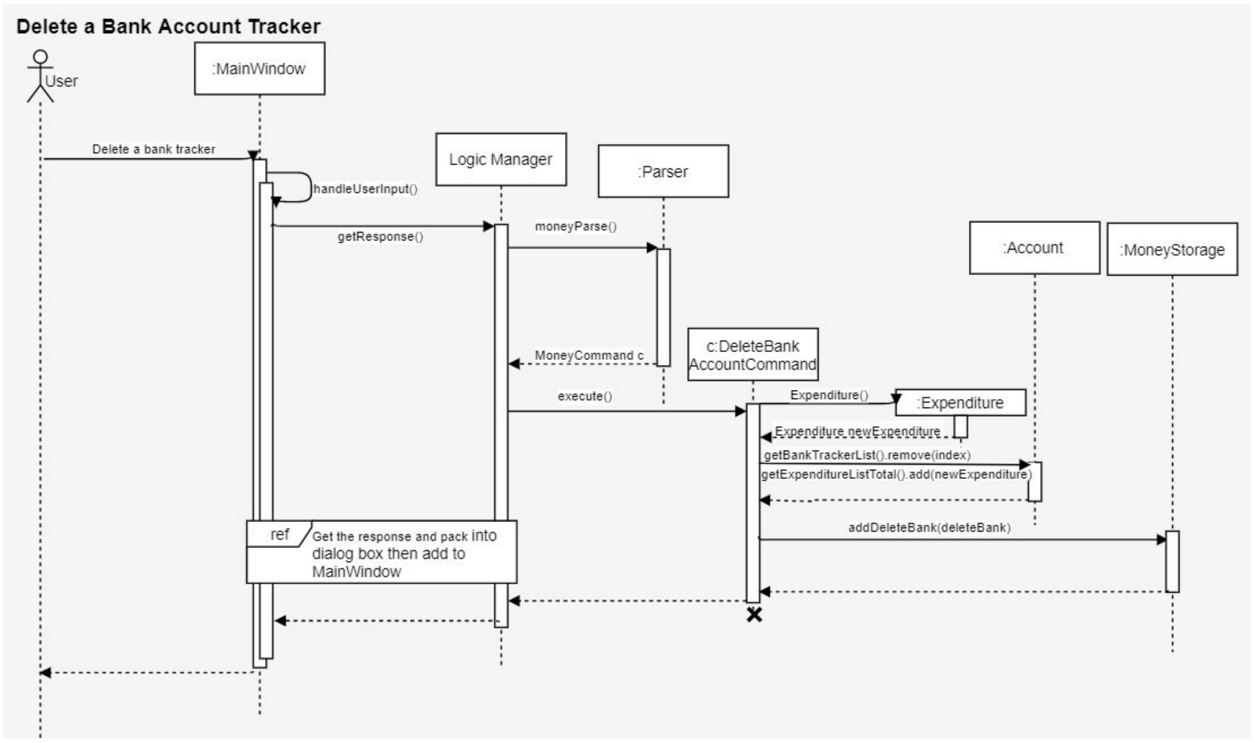
Step1. MainWindow class will handle this command by invoking getResponse method from logic component to get a string of message.

Step2. The input will be parsed and get a DeleteBankAccountCommand, then execute method in DeleteBankAccountCommand will be invoked.

Step3. During invoking the execute method, the bank tracker in the list will be deleted and a new expenditure will be created because of the missing balance of the deleted tracker and added into the overall list.

Step5. Finally, MoneyStorage will store the changes and the details of the deleted tracker will be printed.

The graph below demonstrates the DeleteBankAccountCommand.


Delete a Bank Account Tracker
User
:MainWindow
Delete a bank tracker
handleUserInput()
Logic Manager
:Parser
getResponse()
moneyParse()
:Account
:MoneyStorage
MoneyCommand c
c:DeleteBank AccountCommand
execute()
Expenditure()
:Expenditure
Expenditure newExpenditure
getBankTrackerList().remove(index)
getExpenditureListTotal().add(newExpenditure)
addDeleteBank(deleteBank)
ref
Get the response and pack into dialog box then add to MainWindow

3.6.3. List Bank Trackers

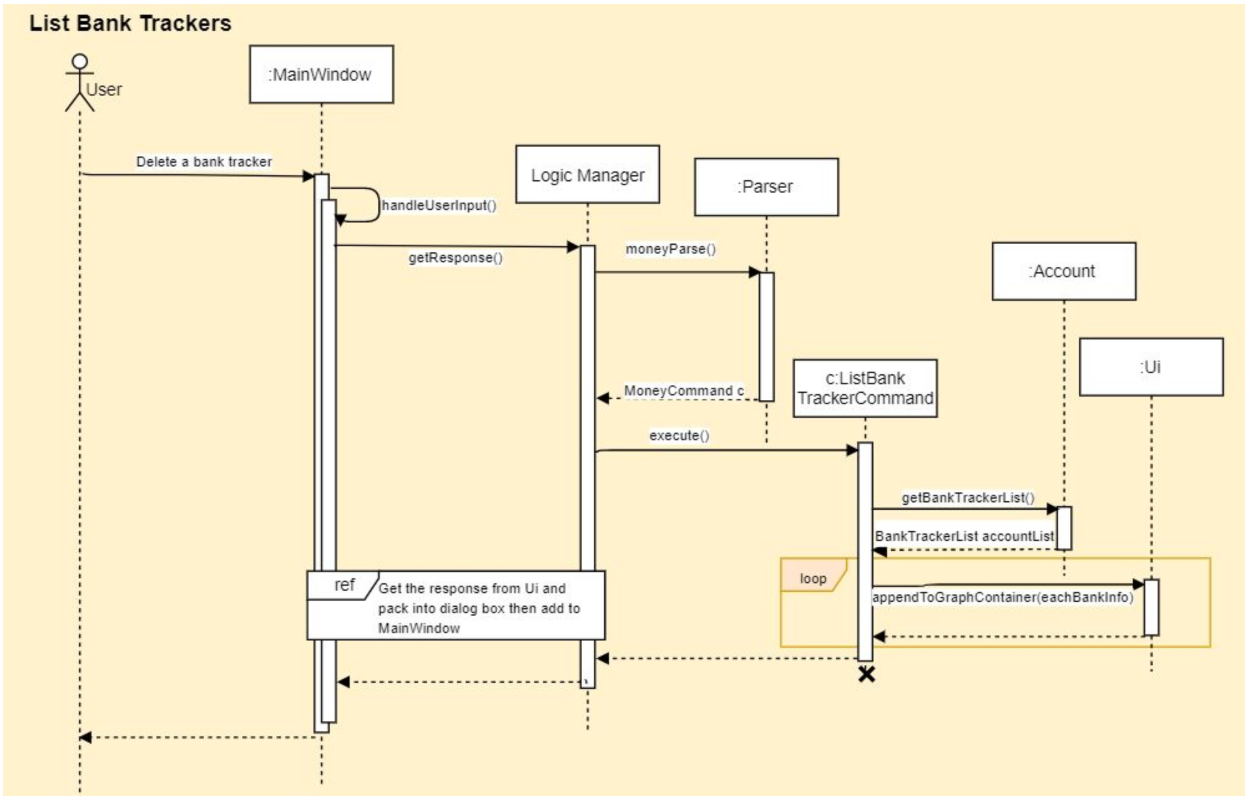
This feature allows users to list down all the bank account trackers.

Step1. The input will be parsed in the logic component and get a ListBankTrackerCommand, then execute method in ListBankTrackerCommand will be invoked.

Step3. During invoking the execute method, all existing bank account trackers will be appended to the output string in Ui by invoking appendToOutput method.

Step4. Finally, the response will be packed into dialog box and presented on GUI.

The graph below demonstrates the ListBankTrackerCommand.



3.6.4. Check Future Balance

This feature helps to check the balance of an account on a specific date.

Step1. User type in command to check the future balance of an account, which is handled by handleUserInput method in MainWindow.

Step2. MainWindow class will handle this command by invoking getResponse method from logic manager to get a string of message.

Step3. The logic manager will parse the user input then get the CheckFutureBalanceCommand to execute. During the execution, the bank account the user would like to check will be found by findTrackerByName method in Account class. Then the predictAmt method in that BankTracker will be invoked to get the future balance.

Step4. Finally, the information will be appended to the output string in Ui which will be presented by MainWindow class

3.6.5. Internal Transfer (Withdraw/Deposit)

This feature helps to track the income and expenditure from the bank account. When withdraw/deposit the money, a new expenditure/income will be created.

Step1. User types in the command which will be handled by MainWindow and sent to logic component to get parsed.

Step2. After the parsing process,execute method in InternalTransferCommand class will be invoked and withdraw or deposit money to the account tracker respectively.

Step3. After changing the account, a new income/expenditure will be recorded to the total income/expenditure list.

Step4. Finally, the updated details of the changed bank account tracker will be shown on the left hand side pane.

3.7. Search Feature

This feature allows the user to search for items stored in the program based on keyword/keywords entered.

3.7.1 Current Implementation

The find mechanism is facilitated by the FindCommand from the Logic Component. The command is called every time the user types into the search bar in the application.

Given below is an example usage scenario and how the find mechanism behaves at every step.

Step 1: The user types the keyword/keywords into the search bar in the application.

Step 2: Upon key release of the character typed into the search bar, the Logic Manager (FG) takes the current input in the search bar and calls moneyParse on Parser with that input.

Step 3: Parser is called and returns a FindCommand object to the Logic manager (FG).

Step 4: The Logic Manager (FG) will call execute() on the FindCommand object.

Step 5: The Logic component will interact with the Account model component to search for items with descriptions which matches the keyword/keywords.

Step 6: The command result will return the search results split according to the different types of items (e.g goals, income, expenditure etc.) as a string and the results will be displayed in the GUI.

3.7.2 Design Considerations

Aspect: Displays the real time search results as the user types in the keywords into the search bar.

- Alternative 1 (current choice): Passes contents of the search bar on every key release.
  - Pros: Easy to implement
  - Cons: May have performance issues as the number of commands executed is large if the search input is large.
- Alternative 2: Detect when the user stops typing the entire search phrase.
  - Pros: Reduces the number of commands executed by the system, avoiding performance issues.
  - Cons: Difficult to determine when the user stops typing the entire phrase as each person has varying typing speeds and patterns.

3.8. Timing Shortcuts Feature

This feature allows users to enter a shortcuts in place of the full format for dates: `d/m/yyyy` when inputting commands involving dates.

Shortcut dates entered are dates relative to the current date that vary depending on the user input.

Timing shortcuts implemented thus far:

- `now` (Shortcut for the current date)
- `ytd` (Shortcut for yesterday's date)
- `tmr` (Shortcut for tomorrow's date)
- `lstwk` (Shortcut for last week's date)
- `nxtwk` (Shortcut for next week's date)
- `lstmth` (Shortcut for last month's date)
- `nxtmth` (Shortcut for next month's date)
- `lstyr` (Shortcut for last year's date)
- `nxtyr` (Shortcut for next year's date)

3.8.1 Current Implementation

The shortcuts are implemented by a series of case statements in the method, `shortcutTime()`, within the `Parser` component.

A command calls the method `shortcutTime()` on a user given date input from `Parser`. The code makes use of the Calendar API's ability to add and subtract days, weeks, months or years to a given date. The result is then parsed to a `LocalDate` object and returned to the command.

3.8.2. Design Considerations

- Alternative 1: Shortcuts are not abbreviated
  - Pros: More user friendly
  - Cons: Less convenient and efficient
- Alternative 2 (current implementation): Shortcuts are abbreviated
  - Pros: More efficient
  - Cons: Harder to understand

3.9. Help Feature

This feature allows users to manage and set their instalments.

It comprises of the following functions:

1. Autocomplete function
2. History function

3.9.1. AutoComplete Function

The AutoComplete Function is facilitated by the MainWindow from the UI Component. A dynamic drop down list is created in this process.

The dynamic drop-down list is created using **org.controlsfx.control.textfield.TextFields** API. The **bindAutoCompletion** method in the **TextFields** creates a new auto-completion binding between the given textField and the given suggestion provider.

Possible commands are checked against the user input and added to the TreeSet **suggestedCommands** accordingly, which will be displayed as the drop-down list.

A command is added if the following conditions are satisfied:

1. The user input is not empty. This is to prevent the list from dropping down every single time when the command bar is cleared.
2. The user input is a valid command. There is no need for any suggestions if the user keys in a command that matches with one of the correct commands
3. The command starts with the user input. This is so as to suggest the correct user command that the user is trying to type.

Given below is an example usage scenario and how the AutoComplete mechanism behaves at every step.

Step 1: The user types in the first letter of the commands that they want to type into the command bar.

Step 2: Upon key release of the character typed into the command bar, the Logic Manager (FG) calls autoCompleteFunction and an AutoComplete object is created. A TreeSet of suggested commands is created at the same time.

Step 3: AutoComplete object will iterate through the library of commands and push the match commands that satisfied the three conditions above into the TreeSet of suggested Commands.

Step 4: The TreeSet of suggested commands will then be returned and will be bind with the TextField UserInput.

Step 5: As user continues to key in more letter into the command bar, Step 2 to 4 is repeated over and over again until the user choose to select a command as mentioned in Condition 2 above.

<u>**3.9.2.History Function**</u>

The History Commands Functions are facilitated by the MainWindow from the UI Component. A History Object is created at the initialization of the program. The function in the MainWindow acts on `↑` and `↓` arrow keys.

Given below is an example usage scenario and how the History mechanism behaves at every step.

Step 1: The user types a string into the command bar and press enter and the string will then be parsed Logic Manager to be processed. At the same time, the string will be pushed into the list.

Step 2: Step 1 is repeated as the user continues to type in commands and press enter.

Step 3: User then press the `↑` arrow key on the keyboard.

Step 4: Upon key released of the `↑` arrow key, this Keyevent is captured and handled by the help() function in MainWindow.

Step 5: The History Object will iterate through the list of the string that the user has previously typed and entered. The string is returned to TextField UserInput and is shown on the command bar in the GUI. The index will be set as well.

Step 6: Step 3 to 4 can be repeated as long as the boundary is not reached. The boundaries are the first string that the user has typed on initialization of the program and the last string that the user has typed. The `↓` arrow key work in the same way as the `↑` arrow key, but in the opposite direction.

3.9.3. Design Considerations

- **Auto-Completion Binding**

Alternative 1: Only suggests Commands that starts with user input (Current Choice)

Pros: Suggested Commands are accurate.

Alternative 2: Suggesting any commands that contains user input

Pros: Displays more suggestions.

Cons: Unnecessary and Inaccurate suggestions tends to appear.

- **Data Structure of suggestedCommands in Autocompletion**

TreeSet is being used. TreeSet seems to be the most feasible data structure as the list of suggestedCommands needs to be sorted and TreeSet's basic operation have a time complexity of O(logn) time which is decently fast. This allows the binding to take place fast and allows the program to give almost immediate suggestions to the user as the user types.

- **Disabling History function when Autocompletion happens**

As both features of the help function acts on the same TextField UserInput (Command bar), KeyEvent that can be captured by both feature. The list of suggestedCommands can be navigated with the `↑`, `↓`, `and enter keys`. However, these three key events can also be captured by the History function or the command bar itself. Hence, it is necessary to disable them when the dynamic drop down list of suggested commands is being thrown.

3.10. Statistics Feature

3.10.1 Current Implementation

Statistics feature provides the user with some graphs like histograms, as an illustration, to know the trend or the status of monthly income and expenditure clearly. To show the user the graph, data will be passed through `DataTransfer` (an interface) accordingly from `Account` depending on the type of information. Then the data will be processed by `Histogram` or `LineGraph` to get the formatted graph. Finally, the graph will be sent to `MainWindow` which is the general GUI window's controller to present the graph to the user. Besides, `GraphCommand` will prompt a reply in a dialogue box.

This feature contains 3 types of commands:

- graph monthly report - Get the histogram of current month's income and expenditure
- graph income/expenditure trend – Get the line chart for overall income or expenditure's trend based on its types
- graph finance status /until [date] – Get the histogram for 3 months income and expenditure before the given date.

This feature includes histogram and line chart as an illustration. All graph commands' mechanisms behave in a similar way which will be shown below:

Step 1. The user sends the graph command to the text box. The input string will be handled by MainWindow first. GraphContainer will add the corresponding graph based on the type of the input string by calling the function from DataTransfer.

eg. input: graph monthly report, DataTransfer.getMonthlyData(duke.getAccount) will be called to return the histogram to add into the graph container.

Step 2. The function in DataTransfer will get data from Account and pass the data to different GUI controllers based on the type of input string. Then it will transfer the graph back to graph container in MainWindow class.

eg. input: graph monthly report, Histogram.getHistogram() will be invoked to return the graph with data.

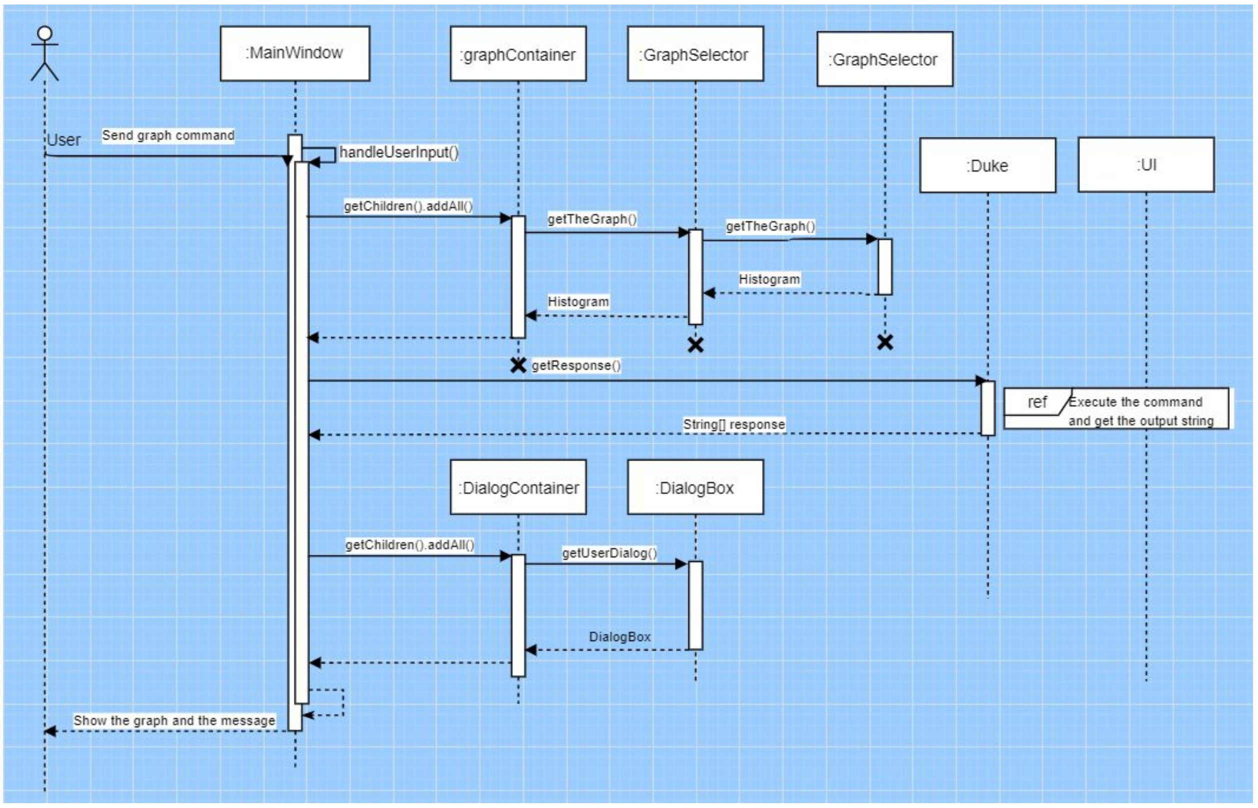
Step 3. After presenting the graph, the input string will be handled by Duke which will return a string as a response to the dialogue container in MainWindow.

Step 4. Duke will pass the graph command to Parser, then Parser will identify the input string and return a graph command to Duke.

Step 5. Duke will execute the command from Parser and the Response will be appended to the output string in Ui.

Step 6. Finally, the output string in Ui will be returned to the dialogue box in MainWindow and added to the dialogue container and presented to the user.

The sequential diagram below illustrates the process of statistics feature.



3.10.2 Design Considerations

Aspect: Let the user choose the types of graphs or present the data with the fixed type of graph?

- **Alternative 1 : Present the data with fixed graphs' type.**

Pros: Easy to implement, requires less judgement and make the command user-friendly. Since we show the graph based on the property of the data (eg. show line chart for income/expenditure trend)

Cons: Can only show one type of graph for one type of data. (eg. can only show a histogram for monthly income and expenditure report)

- **Alternative 2 (current choice): User can choose their desired graph type.**

Pros: Give the user more choice and present the data in different ways.

Cons: Make code messy since it needs some more judgements. User needs to remember more commands to type in.

3.11. Undo Feature

This feature allows the user to undo the effects of the last 5 commands issued to the program, The commands that can be undone include:

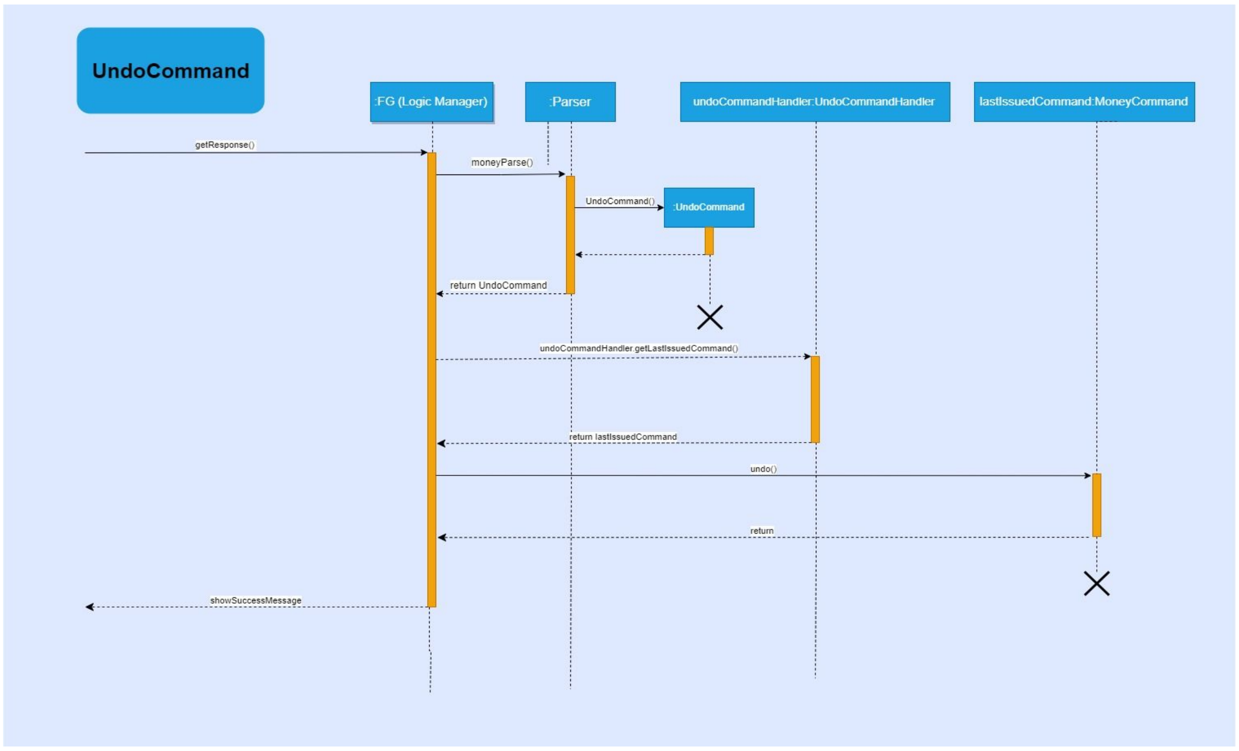
- Any command that *Adds* an Item to the Account class
- Any command that *Deletes* an Item from the Account class
- Any command that involves *changing properties of* items in the database

3.11.1 Current Implementation

The following shows a step by step analysis of how the undo command is executed.

1. In the LogicManager, there is an *undoCommandHandler* with a lastIssuedCommands stack that records the last 5 commands issued to the program (excluding undo command itself).
2. User wishes to undo the last command issued by typing in the command “undo”, which is passed into moneyParse.
3. moneyParse returns a MoneyCommand of type UndoCommand to LogicManager.
4. LogicManager retrieves the last command issued to the stack in *undoCommandHandler*. That command is popped from the stack.
5. LogicManager calls undo() of lastIssuedCommand to revert program to previous state.
   a. If lastIssuedCommand is an Add command:
      i. undo() gets the list that the respective item is in and removes the last item on that list.
      ii. writeToFile() is called to update the data in local hard disk.
      iii. UI prints a success message.
   b. If lastIssuedCommand is a Delete command:
      i. The entry deleted by the command is stored in a stack that is saved in MoneyStorage.
      ii. undo() retrieves the deleted entry from the stack in MoneyStorage.
      iii. The deleted entry is inserted in its original place.
      iv. writeToFile() is called to update data in local hard disk.
      v. UI prints a success message.
   c. If lastIssuedCommand is neither an Add or Delete command:
      i. UI prints an error message.
6. lastIssuedCommand gets updated, and FG continues.

A brief illustration of this process can be found in the sequence diagram below:



3.11.2 Design Considerations

- **Undoing Multiple Commands**

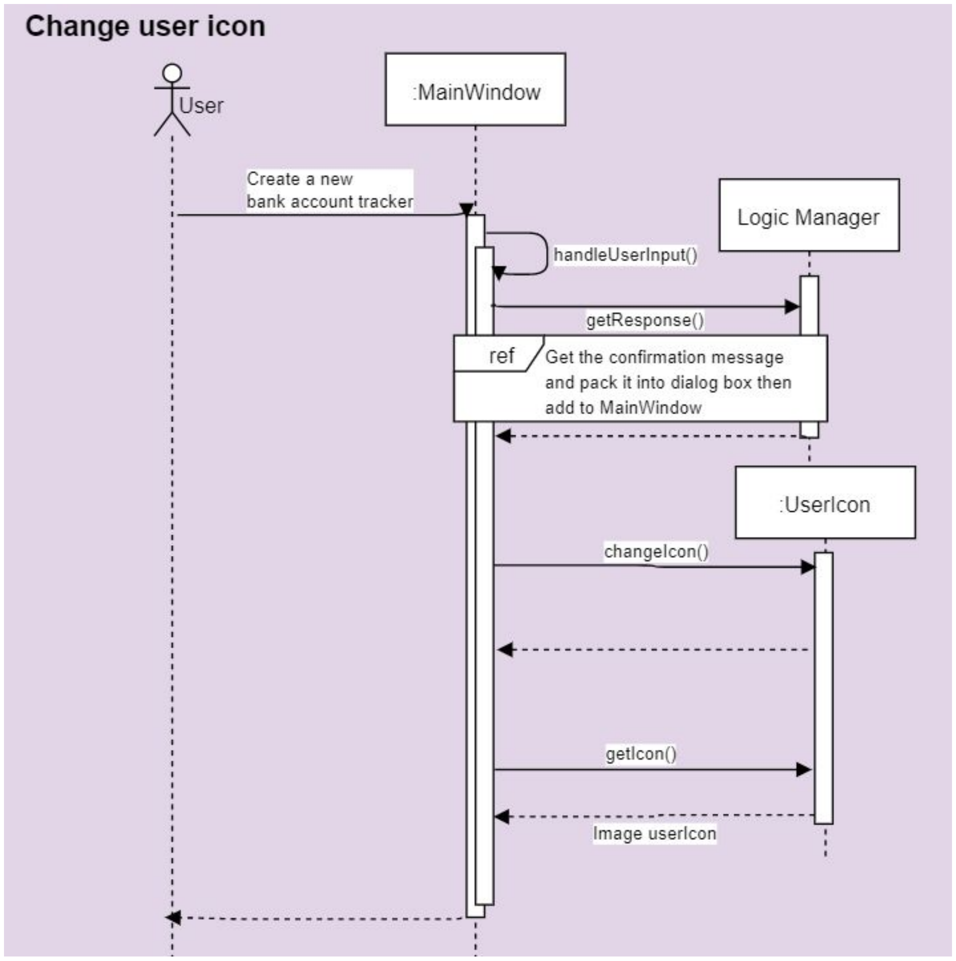
The program can undo up to 5 successive commands issued to the program. The order of the commands is preserved as only commands that delete entries from the account will update the stack in MoneyStorage.

- **Undoing Undo Commands**

This program at its current implementation is unable to undo the undo commands. Commands of type UndoCommand are not included in the lastCommandsIssued stack in UndoCommandHandler.

3.12. Change Icon

This feature allows the user to choose and store their own user icon.



Step1. A user icon will be initialized by the constructor in UserIcon once the user run the overall program.

Step2. When the user try changing icon, the command will be handled by the MainWindow and invokes the changeIcon method in UserIcon, which opens a file selector for the user to choose the icon.

Step3. During the changeIcon method, the icon’s image will be copied to the folder “dataFG/userCustomizedIcons” and the path will be updated in iconPath.txt.

Appendix A: Product Scope

Target user profile

- Has a need to manage their finances
- Wants to make and keep track of their financial goals
- Prefer working on desktop over other
- Can type fast
- Is comfortable using CLI apps

Value Proposition

- Finance tracker in the short term and long term all in one GUI driven application.
- Able to track things such as income and expenditure, loans, goals and instalments.

Appendix B: User Stories

Priority Level	As a...	I want to...	So I can...
High	Fresh Graduate	Have security and privacy	Protect my sensitive financial information
	Fresh Graduate	Track my savings	Afford housing, cars or other big-ticket items
	Fresh Graduate	Record my expenditure	Live within my own means as I work towards my financial goals
	Fresh Graduate	Track my bank account	Know and manage my bank accounts conveniently
Medium	Fresh Graduate	Track my progress towards my budget goals	Continue to work towards it over time
	Fresh Graduate	Record my debt	Make a more

			comprehensive financial plan and set budget goals to clear it
	Fresh Graduate	Record my investment portfolio	Account for passive income
Low	Fresh Graduate	Calculate my disposable income	Make a better budget plan
	Fresh Graduate	Plot the graph of the income and expenditure	Check the trend of the financial issues directly and clearly
	Fresh Graduate	Be reminded to pay my bills	Pay them on time
	Fresh Graduate	Track my loans to others	Remember to claim from others what is owed to me

Appendix C: Use cases

A use case describes an interaction between the user and the system for a specific functionality of the system.

Use cases capture the **functional requirements** of a system.

Use Case 1

- System: Financial Ghost(FG)
- Actor: User
- Use Case: Check account balance
    1. User opens the application
    2. FG greets
    3. User enters the command to check the current balance
    4. UI prompts the related information about the balance
    5. FG waits for the user’s next command

  Use Case Ends.

- Extensions
    1. The given command is invalid

        - FG displays an error message
        - Use Case continues at step 3

Use Case 2

- System: Financial Ghost(FG)
- Actor: User
- Use Case: Record Finances
    1. User opens the application
    2. FG greets
    3. User enters the command to declare the amount of money as income/expenditure
    4. UI prompts a "successfully recorded" message
    5. FG waits for the user's next command

  Use Case Ends.

- Extensions
    1. The given command is invalid
        - FG displays error message
        - Use Case continues at step 3
    2. The given income has invalid characters in it
        - FG displays error message
        - Use Case continues at step 3

Use Case 3

- System: Financial Ghost(FG)
- Actor: User
- Use Case: View Financial Statistics
    1. User opens the application
    2. FG greets
    3. User enters the command to view the financial statistics
    4. UI displays the relevant statistics to the user
    5. FG waits for the user's next command

  Use Case Ends.

- Extensions
    1. The given command is invalid
        - FG displays an error message
        - Use Case continues at step 3

Use Case 4

- System: Financial Ghost(FG)
- Actor: User
- Use Case: Set Financial Goals
    1. User opens the application
    2. FG greets
    3. User enters the command to add a financial goal into the system
    4. UI displays a “successfully added” message.
    5. FG waits for the user’s next command

  Use Case Ends.

- Extensions
    1. The given command is invalid
        - FG displays an error message
        - Use Case continues at step 3

Use Case 5

- System: Financial Ghost(FG)
- Actor: User
- Use Case: Check Goal progress
    1. User opens the application
    2. FG greets
    3. User enters the command to get a progress report of the user’s progress towards their goals
    4. UI displays a detailed description of the action.
    5. FG waits for the user’s next command

  Use Case Ends.

- Extensions
    1. The given command is invalid
        - FG displays an error message
        - Use Case continues at step 3

Use Case 6

- System: Financial Ghost(FG)
- Actor: User
- Use Case: get recommended savings to reach set goals
    1. User opens the application
    2. FG greets

3. User enters the command to get the recommended saving amount to reach their set goals.
4. UI displays the recommended saving amount as well as the current goal saving amount.
5. UI displays a “successfully added” message.
6. FG waits for the user’s next command

Use Case Ends.

- Extensions
    1. The given command is invalid
        - FG displays an error message
        - Use Case continues at step 3

Use Case 7

- System: Financial Ghost(FG)
- Actor: User
- Use Case: Settle Loans with outstanding debt
    1. User opens the application
    2. FG greets
    3. User enters the command to settle a loan that has been taken/given out from/to another party.
    4. FG clears the loan settled and adds to expenditure/income depending on loan being taken/given out from/to another party
    5. UI displays a “loan settled” message.
    6. FG waits for the user’s next command

Use Case Ends.

- Extensions
    1. The given command is invalid
        - FG displays an error message
        - Use Case continues at step 3

Use Case 8

- System: Financial Ghost(FG)
- Actor: User
- Use Case: Add Loan
    1. User opens the application

2. FG greets
3. User enters the command to add a loan that has been taken/given out from/to another party.
4. UI displays a “successfully added” message.
5. FG waits for the user’s next command

Use Case Ends.

- Extensions
  1. The given command is invalid
     - FG displays an error message
     - Use Case continues at step 3

Use Case 9

- System: Financial Ghost(FG)
- Actor: User
- Use Case: Record Instalments
  1. User opens the application
  2. FG greets
  3. User enters the command to add an instalment into the system.
  4. UI displays a “successfully added” message.
  5. FG waits for the user’s next command

Use Case Ends.

- Extensions
  1. The given command is invalid
     - FG displays an error message
     - Use Case continues at step 3

  2. Auto Deductions

     - Monthly deductions occurs at specific date

  3. Undo/Deletion

     - Remove previously added instalments

Use Case 10

- System: Financial Ghost(FG)
- Actor: User
- Use Case: Archive Financial Statements
  1. User opens the application

2. FG greets
3. User enters the command to archive a specified month’s financial statements.
4. UI displays a “successfully archived” message.
5. FG waits for the user’s next command

Use Case Ends.

- Extensions
    1. The given command is invalid
        - FG displays an error message
        - Use Case continues at step 3
    2. FG fails to write to the archive
        - FG displays an error message

Use Case Ends.

Use Case 11

- System: Financial Ghost(FG)
- Actor: User
- Use Case: Provides Assistance to the User
    1. User opens the application
    2. FG greets
    3. User enters the command to get a detailed description of specified action.
    4. UI displays a detailed description of the action.
    5. FG waits for the user’s next command

Use Case Ends.

- Extensions
    1. The given command is invalid
        - FG displays an error message
        - Use Case continues at step 3

Use Case 12

- System: Financial Ghost(FG)
- Actor: User
- Use Case: Search for items
    1. User opens the application
    2. FG greets

3. User enters the command to get search for an item using a keyword by specifying the desired field of search.
4. UI displays a list of the search results.
5. FG waits for the user’s next command

Use Case Ends.

- Extensions
    1. The given command is invalid
        - FG displays an error message
        - Use Case continues at step 3

Use Case 13

System: Financial Ghost(FG)

- Actor: User
- Use Case: Check Instalments
    1. User opens the application
    2. FG greets
    3. User enters the command to check the instalments that are in the system.
    4. UI displays all of User’s instalments to the output.
    5. FG waits for the user’s next command.

Use Case Ends.

- Extensions
    1. The given command is invalid
        - FG displays an error message
        - Use Case continues at step 3

Use Case 14

System: Financial Ghost(FG)

- Actor: User
- Use Case: Use Timing Shortcut
    1. User opens the application
    2. FG greets
    3. User enters the command with a date input using a timing shortcut
    4. UI displays a “successfully recorded” message.
    5. FG waits for the user’s next command.

Use Case Ends.

- Extensions
  1. The given command is invalid
     - FG displays an error message
     - Use Case continues at step 3

Use Case 15

- System: Financial Ghost(FG)
- Actor: User
- Use Case: Create a bank account tracker
  1. User opens the application
  2. FG greets
  3. User enters the command to create a bank account tracker
  4. UI displays a “successfully recorded” message.
  5. FG waits for the user’s next command

  Use Case Ends.

- Extensions
  1. The given command is invalid
     - FG displays an error message
     - Use Case continues at step 3
  2. FG fails to write to the archive
     - FG displays an error message

  Use Case Ends.

Use Case 16

- System: Financial Ghost(FG)
- Actor: User
- Use Case: Undo previous command
  1. User opens the application
  2. FG greets
  3. User enters a command affecting their account’s database
  4. User enters the undo command to revert database before the command in Step 5 was implemented
  5. UI displays a message describing which command was undone
  6. FG waits for the user’s next command

`Use Case Ends.`

- `Extensions`
    1. `User inputs undo command when no previous commands affecting database were made in that session`
        - `FG displays error message`

`Use Case Ends.`

Appendix D: Non-functional requirements

Non-functional requirements specify the constraints under which system is developed and operated.

1. Resolution or size of the GUI
2. Information on budget and expenditure should be saved to local hard disk so they can be loaded when the user checks it in another session
3. Being able to load a particular user's information when he/she logs in to the application.
4. Information on local hard disk should only be accessible and editable through the application itself such that users cannot circumvent the app's security
5. Testability of the app
6. Parsing the relevant information from the user's input to process the commands properly
7. Handling invalid inputs and other potential errors

Appendix E: Glossary

1. FG - Financial Ghost: The name of our application
2. UI - User Interface: Things that the user will interact while using the application

Appendix F: Instructions For Manual Testing

1. **Launch and Shutdown**
    a. Initial launch
        i. Download the jar file and copy into an empty folder

   ii. Double click the jar file or key in the following command in the terminal `java -jar {jarName}`

       Expected: Shows the GUI of Financial Ghost. The window size may not be optimum.

2. **Testing of Income and Expenditure Feature**
   a. Add Income
      i. Test case: `add income TA /amt 560 /on 11/10/2019`

         Expected: Income source inputted will be added to the incomeListTotal in Account. Details of the income source will be shown in the status message.

      ii. Invalid date test case: `add income TA /amt 560 /on 1121/10/2019`

         Expected: Error message informing user of an invalid date

      iii. Invalid input test case: `add income TA /amt blah blah blah`

         Expected: Error message informing user of the correct input format

   b. Add Expenditure
      i. Test case: `spent ultra boost /amt 175 /cat shoes /on 1/6/2019`

         Expected: Expenditure inputted will be added to the expListTotal in Account. Details of the expenditure will be shown in the status message.

      ii. Invalid date test case: `spent ultra boost /amt 175 /cat shoes /on 1/666/2019`

         Expected: Error message informing user of an invalid date

      iii. Test case: `spent ultra boost /amt 453 /on 4/11/2019`

Expected: Error message informing user of the correct input format

c. List Income
    i. Test case: `list all income`

       Expected: Displays all Income objects in the incomeListTotal in the pane on the right

d. List Expenditure
    i. Test case: `list all expenditure`

       Expected: Displays all Expenditure objects in the expListTotal in the pane on the right

e. Delete Income
    i. Prerequisite: List all income sources with the `list all income` command. Identify the index of the income source to delete

       Test case: `delete income 1`

       Expected: Details of the income source specified will be shown in the status message. The income source will be deleted from the incomeListTotal in Account.

    ii. Negative index test case: `delete income -3`

       Expected: Error message informing the user that the index given is out of bounds

    iii. Invalid input test case: `delete income blah blah`

       Expected: Error message informing the user to input a numerical value as the index

f. Delete Expenditure

        i. Prerequisite: List all Expenditures with the `list all expenditure` command. Identify the index of the expenditure to delete

           Test case: `delete expenditure 1`

           Expected: Details of the expenditure specified will be shown in the status message. The expenditure will be deleted from the expListTotal in Account

        ii. Negative index test case: `delete expenditure -2`

           Expected: Error message informing the user that the index given is out of bounds

        iii. Invalid input test case: `delete expenditure blah blah nonsense`

           Expected: Error message informing the user to input a numerical value as the index

3. **Testing of Archive Feature**
    a. Checking the income data for a specific month
        i. Test case: `check income 10 2019`

           Expected: Income sources dated to October of 2019 will be displayed to the user in the pane on the right

        ii. Invalid month test case: `check income 0 2019`

           Expected: Error message informing the user to enter a month from 1-12

        iii. Invalid year test case: `check income 10 21`

           Expected: Error message informing the user to enter a year from

1000-9999

b. Checking the expenditure data for a specific month
    i. Test case: `check expenditure 6 2017`

        Expected: Expenditures dated to June of 2017 will be displayed to the user in the pane on the right

    ii. Invalid month test case: `check expenditure -4 2019`

        Expected: Error message informing the user to enter a month from 1-12

    iii. Invalid year test case: `check expenditure 10 988`

        Expected: Error message informing the user to enter a year from 1000-9999

c. Checking the income data for the current month
    i. Test case: `list month income`

        Expected: Displays all Income objects dated to the current month in the pane on the right

d. Checking the expenditure for the current month
    i. Test case: `list month expenditure`

        Expected: Displays all Expenditure objects dated to the current month in the pane on the right

4. **Testing of Goal Setting Feature**
    a. Adding a financial goal to the system
        i. Test case: `goal HDB /amt 100000 /by 15/1/2050 /priority HIGH`

Expected: Entered goal will be added to the goal list. Details of the added goal will be shown in the status message. Updated goal list will be shown in the pane on the right.

ii. Other incorrect add goal commands: The date is not in the format d/m/yyyy. Amount keyed in is not a float or an integer.

b. Deleting a Goal from the goal list

   i. Prerequisites: List all goals using the `list goals` command. Multiple goals in the list.

   ii. Test case: `delete goal 1`

   Expected: The first goal in the goal list will be deleted. Details of the deleted goal will be shown in the status message. The updated goal list will be shown in the pane on the right.

   iii. Test case: `delete goal 0`

   Expected: No goal will be deleted. Error details will be shown in the status message.

   iv. Test case: `delete goal -1`

   Expected: No goal will be deleted. Error details will be shown in the status message.

c. Completing a Goal from the goal list

   i. Prerequisites: List all goals using the `list goals` command. Multiple goals in the list.

   ii. Test case: `done goal 1`

   Expected: The first goal in the goal list will be completed. Details of the completed goal will be shown in the status message. The updated goal list will be shown in the pane on the right.

   If the user does not have enough goal savings the goal will not be completed.

   iii. Test case: `done goal 0`

Expected: No goal will be completed. Error details will be shown in the status message.

iv. Test case: `done goal -1`

Expected: No goal will be completed. Error details will be shown in the status message.

d. Commiting a Goal from the goal list

i. Prerequisites: List all goals using the `list goals` command. Multiple goals in the list. Goal must have a tick beside it, indicating that the user has enough money to complete the goal.

ii. Test case: `commit goal 1`

Expected: The first goal in the goal list will be committed. The simulated goal list will be displayed in the pane on the left. The current goal list will be shown in the pane on the right.

If the user does not have enough goal savings the goal will not be committed

iii. Test case: `done goal 4,6`

Expected: The fourth and sixth goal in the goal list will be committed. The simulated goal list will be displayed in the pane on the left. The current goal list will be shown in the pane on the right.

If the user does not have enough goal savings the goals will not be committed

iv. Test case: `done goal -1`

Expected: No goal will be committed. Error details will be shown in the status message.

5. **Testing of Loan Feature**

a. Adding a loan to the system

i. Add incoming loan test case: `borrowed UOB /amt 40000 /on 8/9/2014`

Expected: Incoming loan inputted will be added to the loans in

Account. Details of the incoming loan will be shown in the status message

ii. Add outgoing loan test case: `lent friend /amt 100 /on 6/3/2016`

    Expected: Outgoing loan inputted will be added to the loans in Account. Details of the outgoing loan will be shown in the status message

iii. Invalid input test case: `borrowed blah blah blah`

    Expected: Error message informing the user of the correct format for adding a loan

iv. Invalid date test case: `lent friend /amt 100 /on 6/3/0000`

    Expected: Error message informing the user of the invalid date

b. Listing loans

   i. List all loans test case: `list all loans`

      Expected: Displays all loans in `loans` in Account in the pane on the right

   ii. List all loans test case: `list incoming loans`

      Expected: Displays all incoming loans in `loans` in Account in the pane on the right

   iii. List all loans test case: `list outgoing loans`

      Expected: Displays all outgoing loans in `loans` in Account in the pane on the right

c. Deleting a loan

        i. Prerequisite: List all loans with the `list all loans` command. Identify the index of the loan to delete

            Test case: `delete loan 1`

            Expected: Details of the loan specified will be shown in the status message. The loan will be deleted from the loans in Account

        ii. Negative index test case: `delete loan -2`

            Expected: Error message informing the user that the index given is out of bounds

        iii. Invalid input test case: `delete loan blah blah nonsense`

            Expected: Error message informing the user to input a numerical value as the index

    d. Settling a loan
        i. Prerequisite: User had previously added an outgoing loan with the description “Lydia” that has yet to be settled

            Settle outgoing loan test case: `received 200 /from Lydia`

            Expected: Displays the amount settled and the leftover amount as well as the status of the loan

        ii. Prerequisite: User had previously added an incoming loan with the description “Rachel” that has yet to be settled

            Settle incoming loan test case: `paid 300 /to Rachel`

            Expected: Displays the amount settled and the leftover amount as well as the status of the loan

iii. Prerequisite: User had previously added an outgoing loan with the description “Lydia” that has yet to be settled

   Settle all outgoing loan test case: `received all /from Lydia`

   Expected: Displays the leftover amount settled and the status of the loan which is now set to settled

iv. Prerequisite: User had previously added an incoming loan with the description “Rachel” that has yet to be settled

   Settle incoming loan test case: `paid all /to Rachel`

   Expected: Displays the leftover amount settled and the status of the loan which is now set to settled

6. **Testing of Instalment Feature**
   a. Adding an instalment to the system
      i. Test case: `add instalment mortgage /amt 100000 /within 200 months /from 12/12/2010 /percentage 6`

         Expected: Entered instalment will be added to the instalment list. Details of the added instalment will be shown in the status message. Updated instalment list will be shown in the pane on the right.

      ii. Other incorrect add instalment commands: The date is not in the format d/m/yyyy. Amount keyed in is not a float or an integer.
   b. Deleting an instalment from the instalments list
      i. Prerequisites: List all goals using the `list all instalments` command. Multiple instalments in the list.

      ii. Test case: `delete instalment 1`

         Expected: The first instalment in the instalments list will be deleted. Details of the deleted instalment will be shown in the status message. The updated instalment list will be shown in the pane on the right.

iii. Test case: `delete instalment 0`

Expected: No instalment will be deleted. Error details will be shown in the status message.

iv. Test case: `delete instalment -1`

Expected: No instalment will be deleted. Error details will be shown in the status message.

7. **Testing of Bank Account Feature**
    a. Adding a bank account tracker
        i. Test case: `bank-account OCBC /amt 0 /at 27/7/2017 /rate 0.005`

        Expected: A new bank account tracker will be added into the bank account tracker list. The details of added bank account tracker will be shown in the left hand side pane.

        ii. Test case: `bank-account OCBC /amt 0 /at 2017 /rate 0.005`

        Expected: An exception will be thrown and the relevant error message will be printed out on the pane due to the incorrect date format.

    b. Delete an existing bank account tracker
        i. Test case: `delete bank-account 1`

        Prerequisite: If there is no existing bank account tracker, this command will invoke an exception due to wrong index number.

        Expected: After satisfying prerequisite, this command will delete the first bank account tracker from the list and print out its details on the left hand side pane.

        ii. Other invalid command format: Wrong index number which is not following the rule, (1 <= index <= size)
    c. List all existing bank account trackers
        i. Test case: `list bank trackers`

Expected: The details of all the bank account trackers are listed on the right hand side pane and a confirmation message will be printed on the left hand side pane.

d. Check the future balance of an existing bank account tracker
    i. Test case: `check-balance OCBC /at 12/3/2020`

       Prerequisite: The bank account tracker with the specific name must exist. Otherwise, an exception will be thrown.

       Expected: The balance of the bank account at the given specific date will be printed out in 2 decimal places

    ii. Test case: `check-balance OCBC /at 2020`

       Expected: Although the bank account tracker with that name exists, this input will still invokes an exception and print the error message due to wrong date format which should be dd/MM/yyyy.

e. Internal transfer (withdraw/deposit)
    i. Test case: `deposit 200 OCBC /at 28/9/2019`

       Prerequisite: The specific bank account tracker exists and the date is after the latest update date of that bank account tracker.

       Expected: After satisfying the prerequisite, 200 dollar should be deposited into this bank account and the updated details will be printed out on the left hand side pane.

    ii. Test case: `withdraw 200000000 OCBC /at 28/9/2030`

       Prerequisite: The late is later than the latest update date and the specific bank account tracker exists but the balance of the bank account is smaller than the input value.

       Expected: Under this circumstance, this input will invoke an exception and an error message will be printed out due to the over withdraw.

    iii. Other invalid command format: The input bank description cannot be found in the bank tracker list (i.e. such bank account tracker does not exist). Negative input number and wrong number format. Wrong date format.

8. **Testing of Search Feature**
    a. Searching for an item in the system.
        i. Prerequisites: Search input must be keyed into the search bar.
        ii. Test case: enter the desired search input

            Expected: Items which has descriptions that contains the search input will be displayed on the pane on the right. The search results are case-sensitive.

9. **Testing of Timing Shortcuts Feature**
    a. Inputting add command with the shortcut time
        i. Test case: `add income TA /amt 1000 /on now`
            Expected: Income source inputted will be added to the incomeListTotal in Account with the current date. Details of the income source will be shown in the status message.

10. **Testing of Help Feature**
    a. Autocomplete feature
        i. Test case: Type the letter `l` into the command bar.

            Expected: A drop down list of suggested commands that contains commands that starts with the letter ‘l’ will be shown.

        ii. Test case: Type the letter `li` into the command bar.

            Expected: A drop down list of suggested commands that contains commands that start with the letters ‘li’ will be shown.

    b. History feature
        i. Test case: Type in and enter `list all instalments` in command bar. Press `↑` arrow key on the keyboard.

            Expected: The command `list all instalments` will appear on the command text bar.

            Continue Test case: Press `↓` arrow key.

            Expected: The command bar will go back to being empty.

11. **Testing of Statistics Feature**

a. Graph of the current month's income and expenditure
    i. Test case: `graph monthly report`

       Expected: The histogram of the current month's income and expenditure will be shown on the right hand side pane and a confirmation message will be shown on the left hand side pane.

    ii. Test case: `graph monthly report pie_chart`

       Expected: The pie chart of the current month's income and expenditure will be shown on the right hand side pane and a confirmation message will be shown on the left hand side pane.

    iii. Other invalid test case: Besides replacing pie_chart with histogram or line_graph, other patterns does not work and the default pattern (histogram) will be shown.

b. Graph of income and expenditure category trend
    i. Test case: `graph expenditure trend histogram`

       Expected: The histogram of the overall expenditure category trend will be shown on the right hand side pane and a confirmation message will be shown on the left hand side pane.

    ii. Other invalid test case: Besides replacing pie_chart with histogram or line_graph, other patterns does not work and the default pattern (line graph) will be shown.

c. Graph of financial status until the given date
    i. Test case: `graph finance status /until 12/12/2019`

       Expected: The two-series histogram containing 3 months income and expenditure until the given date will be shown on the right hand side pane and a confirmation message will be printed out on the left hand side pane.

    ii. Other invalid test case: Wrong input format.

12. Testing of Undo Feature

Prerequisites: There must be 1-5 undoable commands entered by the user (A full list of undoable commands can be found in the user guide)

a. Testing Undo Function

    i. Test case: Commands that affect the database (adding/ deleting entries in the database)

       Expected: UI prints success message “Last command undone: …”.

    ii. Test case: Commands that do not affect the database (lists/ graphs)

       Expected: UI prints error message “Command can’t be undone!”

b. Testing Undo Stack
    i. Test case: Enter 6 commands (of any type) successively and type `undo` 6 times successively.

       Expected: At the 6th command, UI prints “No commands to undo!”

    ii. Test case: Type `undo` at the start of the program.

       Expected: UI prints “No commands to undo!”