Financial Ghost - Developer Guide

By: CS2113T-W12-2 Since: September 2019 Licence: NUS

1. Setting Up
   1.1. Prerequisites
   1.2. Setting up the project in your computer
   1.3. Verifying the setup
   1.4. Configurations to do before writing code
2. Design
   2.1. Architecture
   2.2. UI Component
   2.3. Logic Component
   2.4. Model Component
   2.5. Storage Component
3. Implementation
   3.1. Goal Setting Feature
   3.2. Loan Feature
   3.3. Instalment Feature
   3.4. Bank Account Feature
   3.5. Better Search Feature
   3.6. Friendlier Syntax Feature
   3.7. Help Feature
   3.8. Statistics Feature
   3.9. Undo Feature
   3.10. Logging
   3.11. Configuration
4. Documentation
5. Testing
6. Dev Ops

Appendix A: Product Scope

Appendix B: User Stories

Appendix C: Use cases

Appendix D: Non-functional requirements

Appendix E: Glossary

Appendix F: Instructions For Manual Testing

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 1.8 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 oss-generic 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 UsingCheckstyle.adoc 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

[overall architecture diagram here]

2.2. UI Component

[Class diagram for Model component here]

UI component is launched by launcher.java and it invokes Main class and MainWindow class to manipulate the overall GUI presented to the user. Based on different user input, the MainWindow class will respond in different ways. Moreover, the feature “auto-complete” has been implemented in order to make typing command more user-friendly. The response can be mainly divided into 3 types.

1. The text response only. This type of input will be handled by handleUserInput method and autoCompleteFunction. In handleUserInput, class Duke will be invoked and proceed to logic part get the corresponding output string presented to the user.

2. The search bar. This type is implemented as a search bar above the main command input box. Type in the keywords and the relevant goals 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 handleUserInput method in MainWindow class and the text confirmation will be handled by Duke.

2.3. Logic Component

[Class diagram for Logic Component here]

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.

Given Below is the sequence diagram for the interactions within the Logic component when a delete command is executed.

[Sequence Diagram for delete income here]

2.4. Model Component

[Class diagram for Model component here]
Structure of the model component is shown in the above 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

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

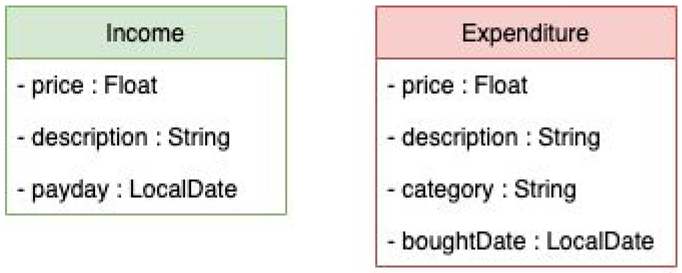
3.1.1. Add Commands Function (Add Income & Add Expenditure)

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.

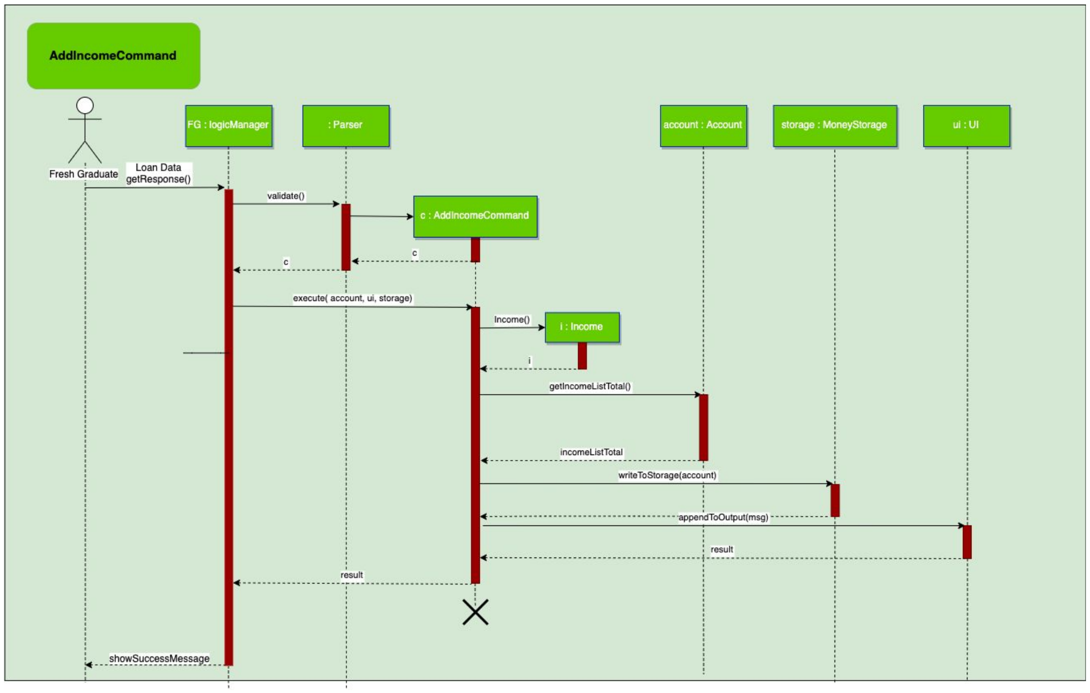
Step 4: The LogicManager calls execute() on the `AddIncomeCommand/AddExpenditureCommand` object.

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 DeleteExpenditureCommand 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.

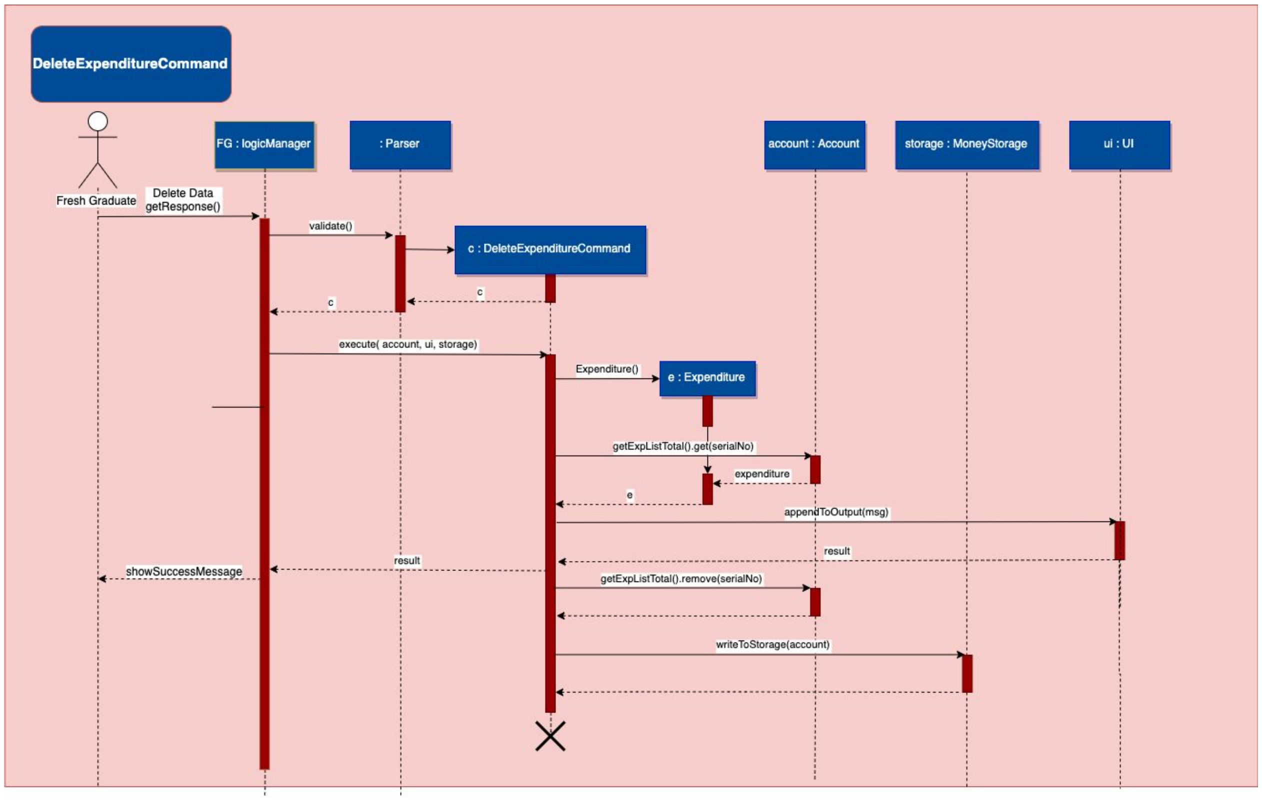
Step 4: The LogicManager calls execute() on the `DeleteIncomeCommand/DeleteExpenditureCommand` object.

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

3.1.1. Current Implementation (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.

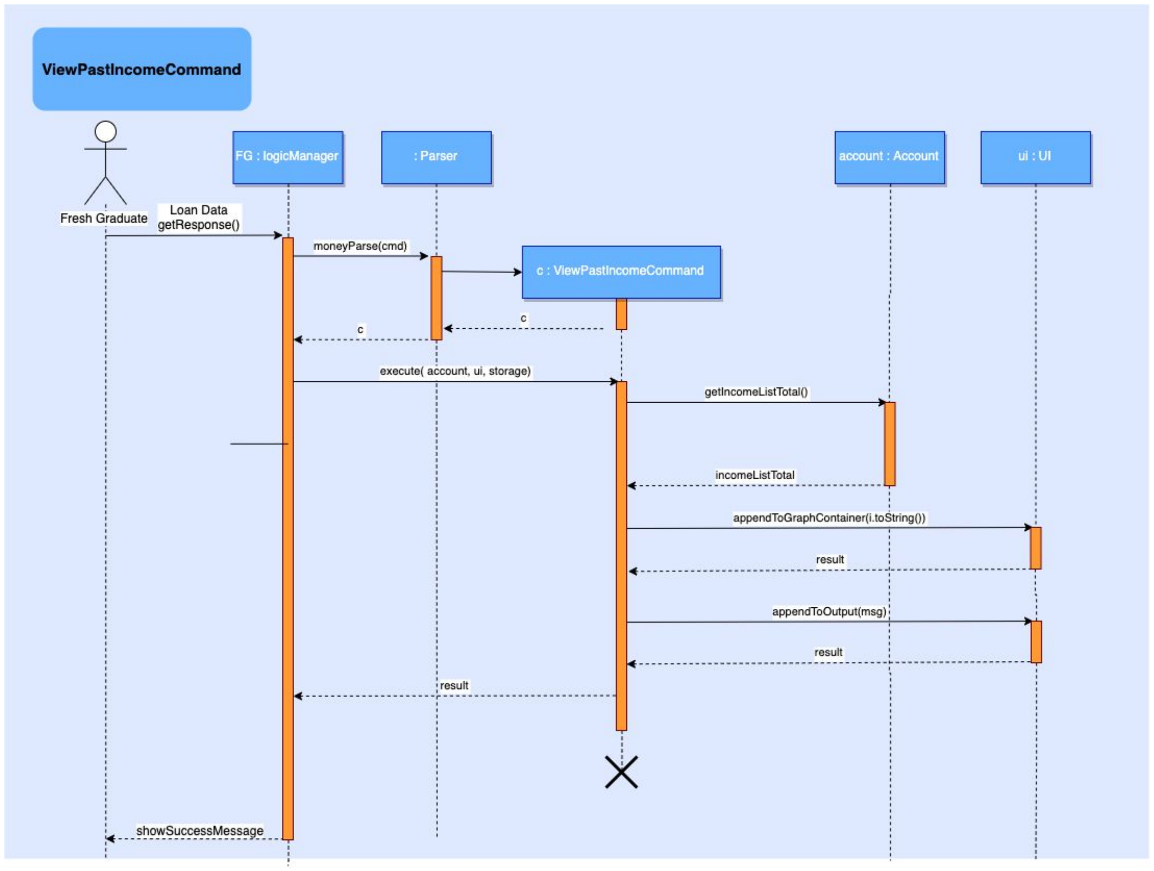
Step 4. The LogicManager calls method execute() on the object.

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`

3.1.4. Current Implementation (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.

3.3. 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(?)

1. Add Goal Function

1.1. Current Implementation

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.

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.

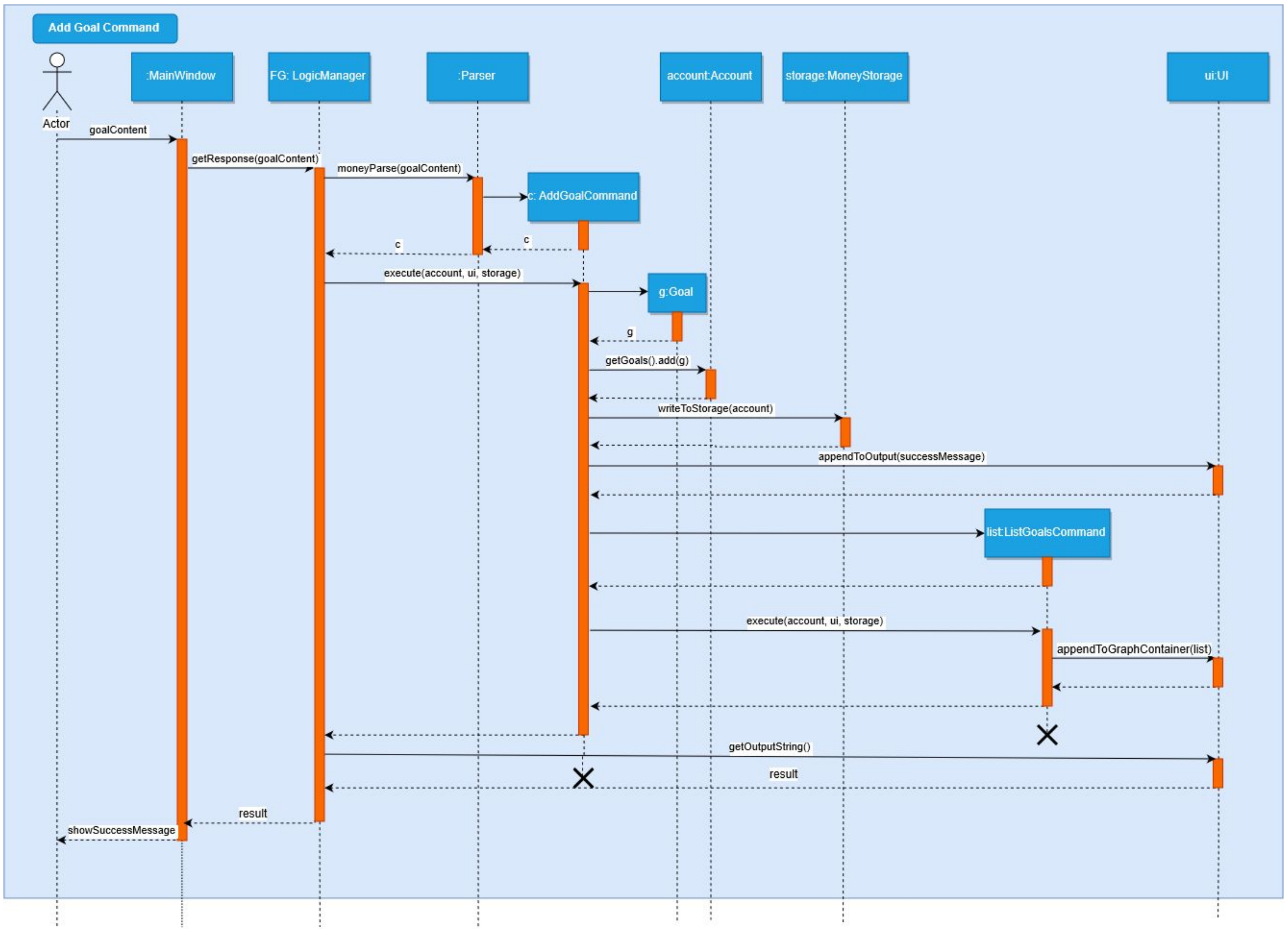
Step 8: ListGoalsCommand will call appendToOutput a Success Message and call appendToGraphContainer the updated Goal List.

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

1.2. Design Considerations

Aspect: Displaying an updated list to the user every time a goal is added to provide the user a clear visual of the goals that he/she has set out for themselves.

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



2. DoneGoal Function

2.1. Current Implementation

The Complete 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 Complete Goal Function behaves at each step.

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

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

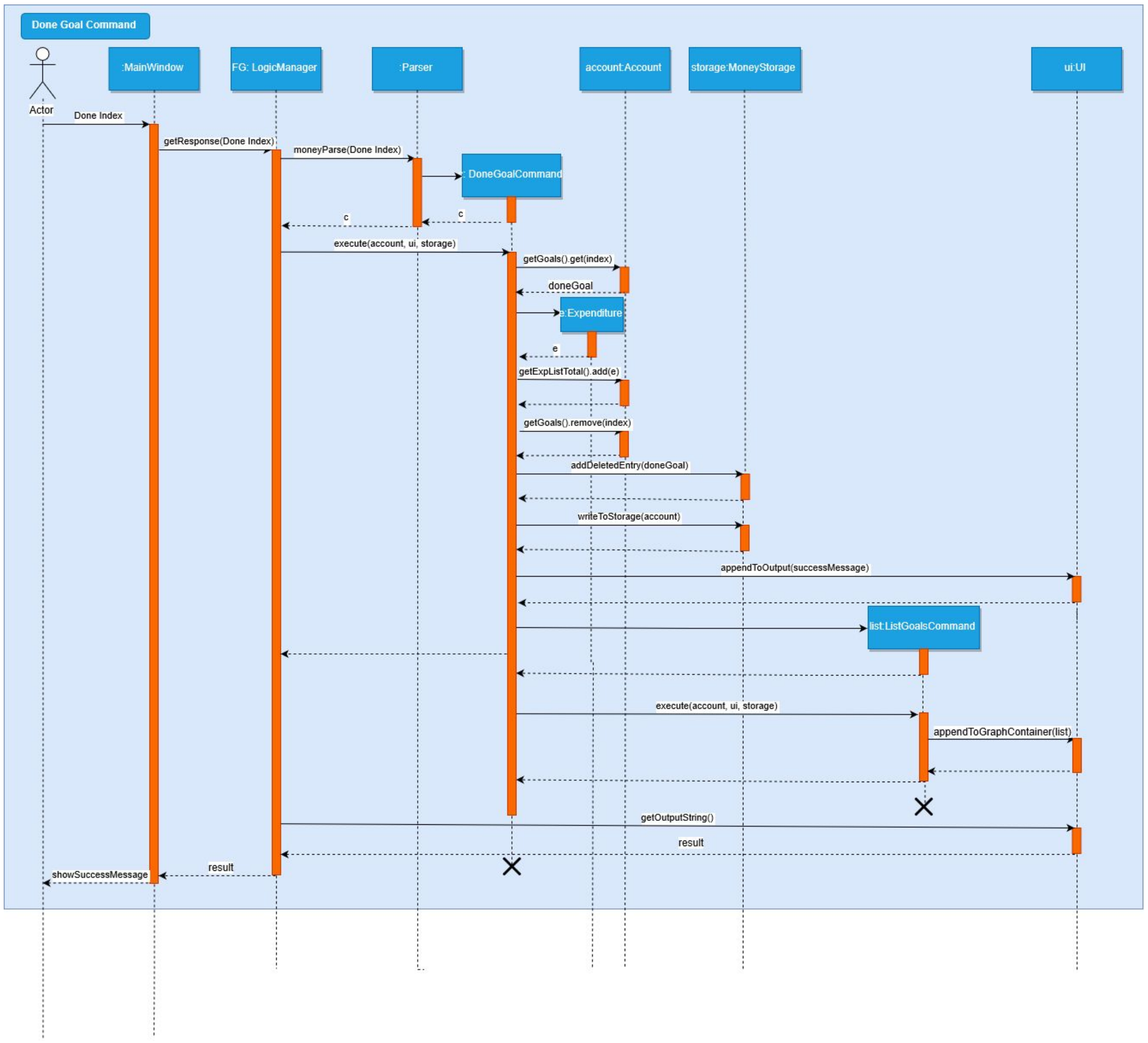
Step 3: Parser is called and constructs a DoneGoalCommand object and returns it to Logic Manager(FG).

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:



2.2. Design Considerations

[need to add design considerations here]

3. Commit Goal Function

3.1. Current Implementation

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”.

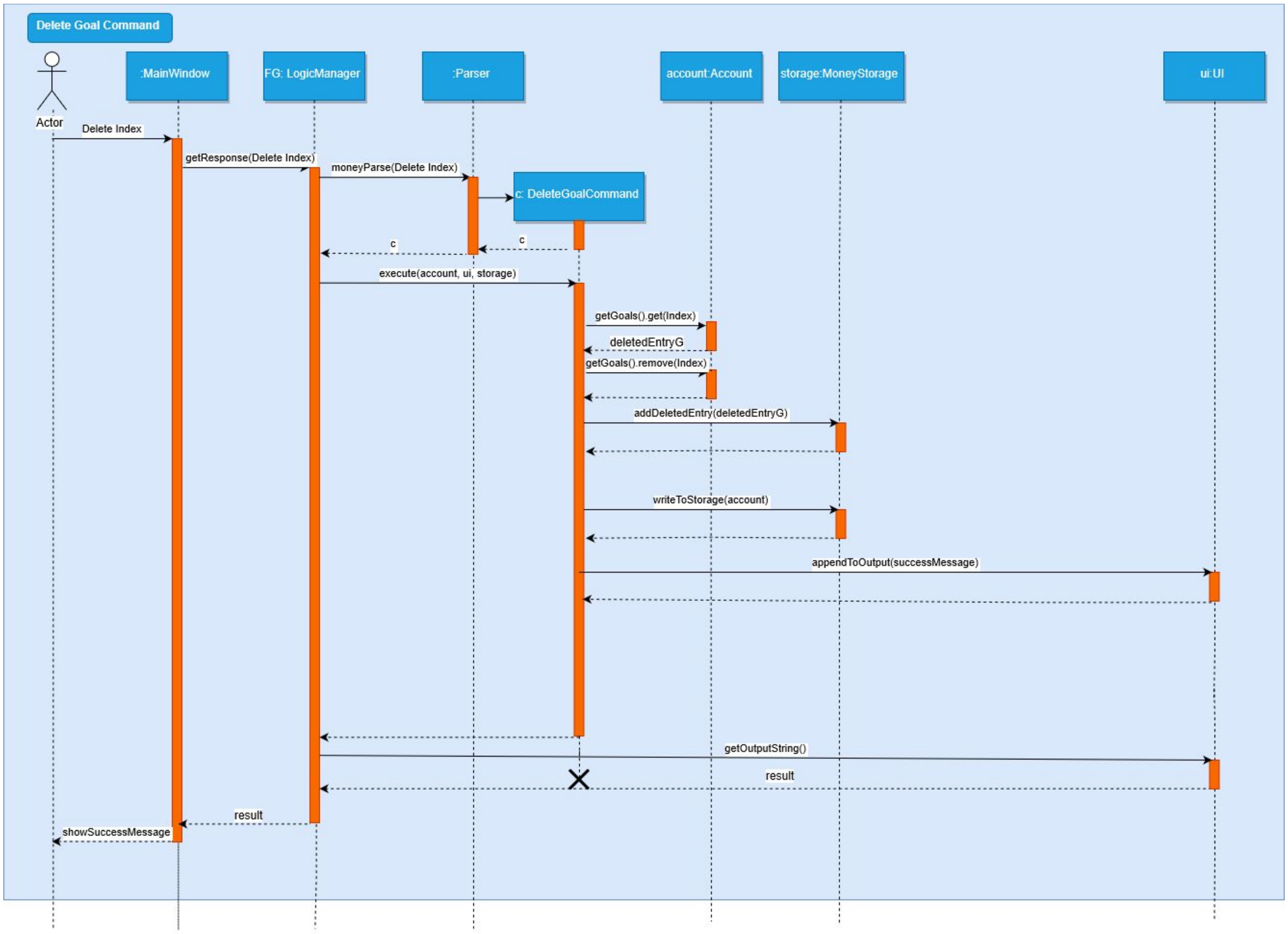
Step 5: CommitGoalCommand will compute the updated finances of the user and the items left in the Goal List after the commit.

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

3.2. Design Considerations

[need to add design considerations here]

4. Delete Goal Function



5. **List Goal Function**

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:

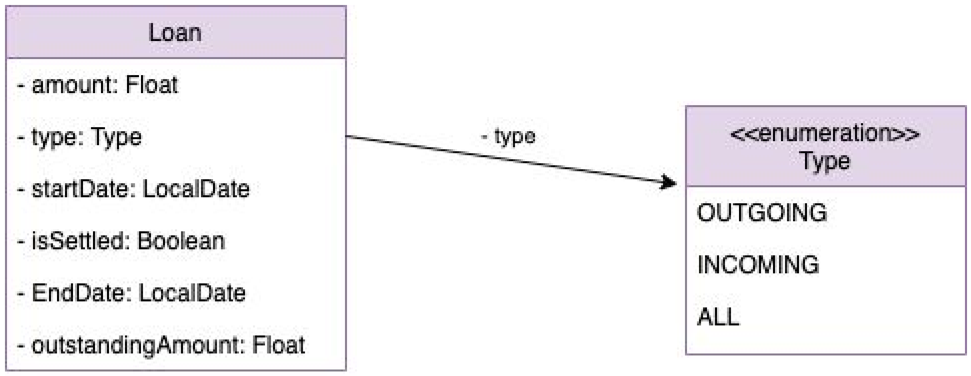
1. Add incoming/outgoing loan
2. List all/incoming/outgoing loans
3. Delete incoming/outgoing loan

4. Settle incoming/outgoing loan

3.2.1 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.

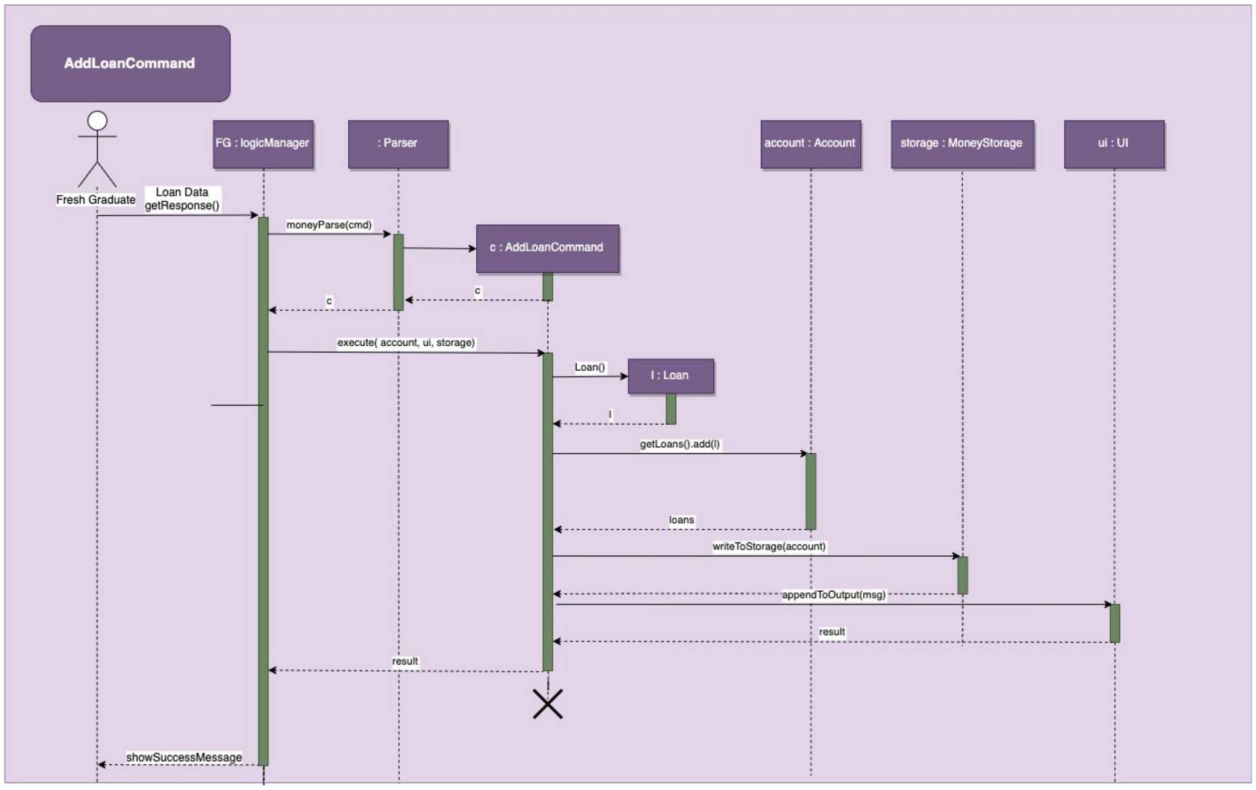
Step 3. Parser is called and returns a AddLoanCommand object to LogicManager.

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



3.2.2 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)

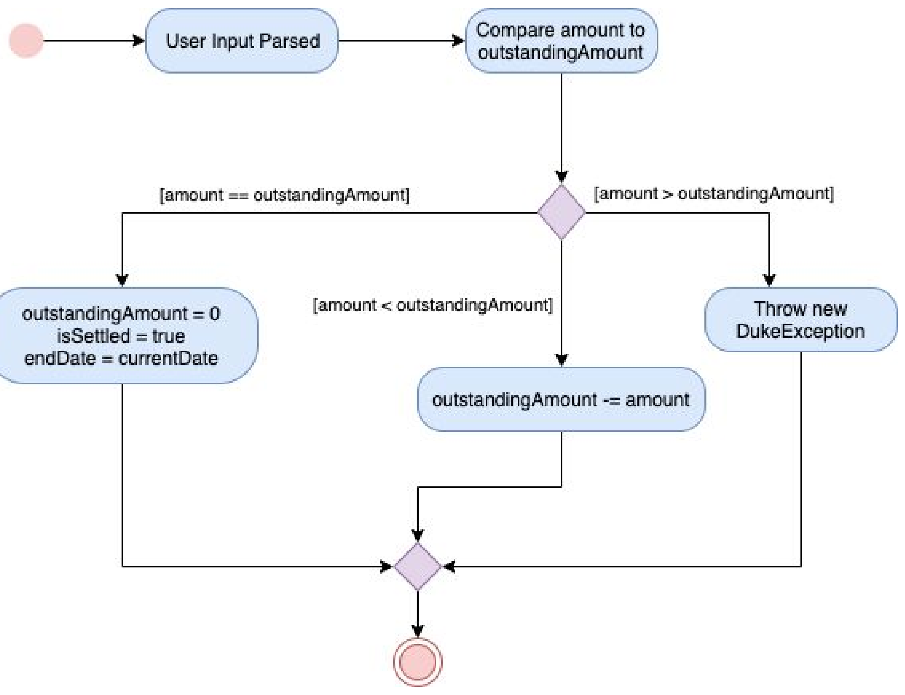
Note: User can choose to input the name or index of the loan party in the Incoming/Outgoing Loan List

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.

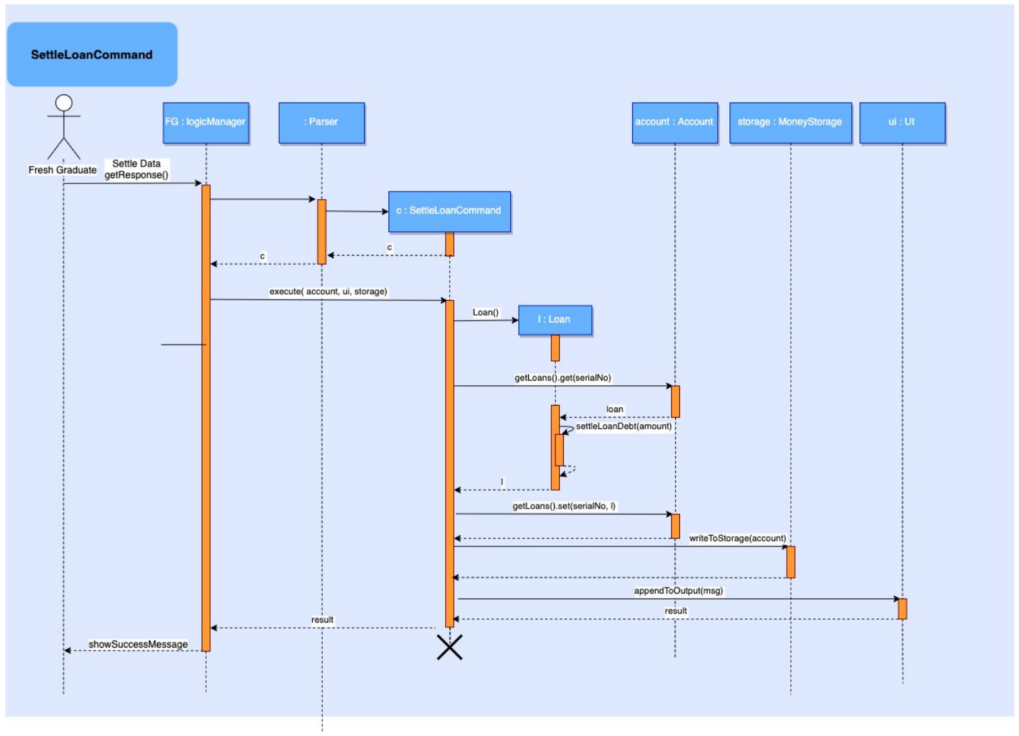
Step 5. The SettleLoanCommand object interacts with Model and searches for the Loan object specified in the user input in the ArrayList of Loans. The `settleLoanDebt` method is called on the Loan object found with the amount in user input used as an argument to settle the debt.
The diagram below shows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

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.



3.2.3 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.

3.2.4 Design Considerations

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. List Instalments Function
4. Delete Instalment Function

3.3.1. 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.

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 @6%

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

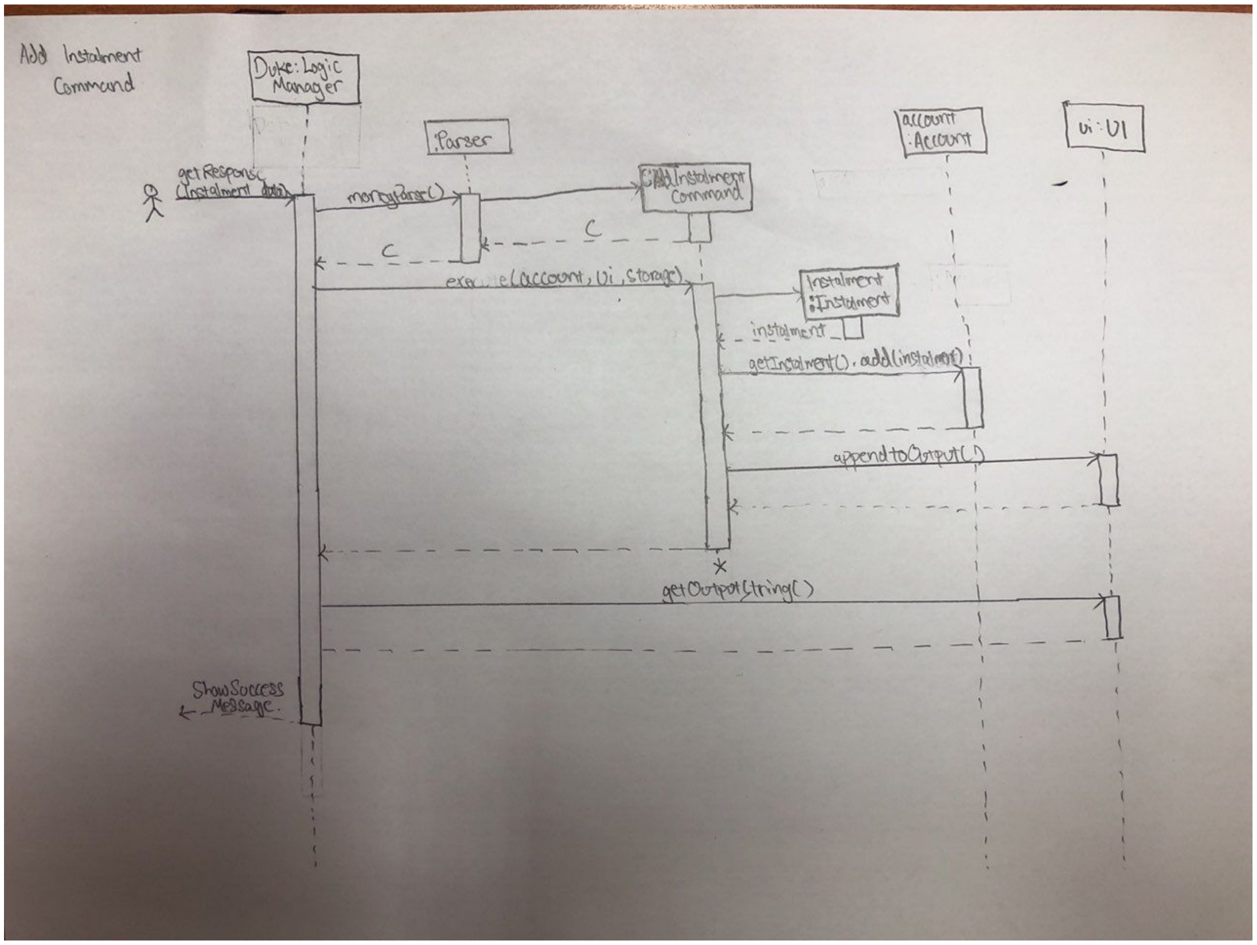
Step 3: Parser is called and it constructs a AddInstalmentCommand object. The AddInstalmentCommand Object is returned to LogicManager.

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.



3.3.2. 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 them

Step 4: If the current date is equal to the pay date of a particular instalment object, 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: AutoUpdateInstalmentCommand will call appendToOutput a Message to inform the user.

[**Insert Sequence Diagram**]

3.3.3. List Instalment Function

3.3.4. Delete Instalment Function

3.3.5. 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:

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

3.7. Better search Feature

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

3.6.1 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.

3.6.2 Design Considerations

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

3.8. Friendlier Syntax Feature

3.9. Help Feature

This feature allows users to manage and set their instalments.

It comprises of the following functions:

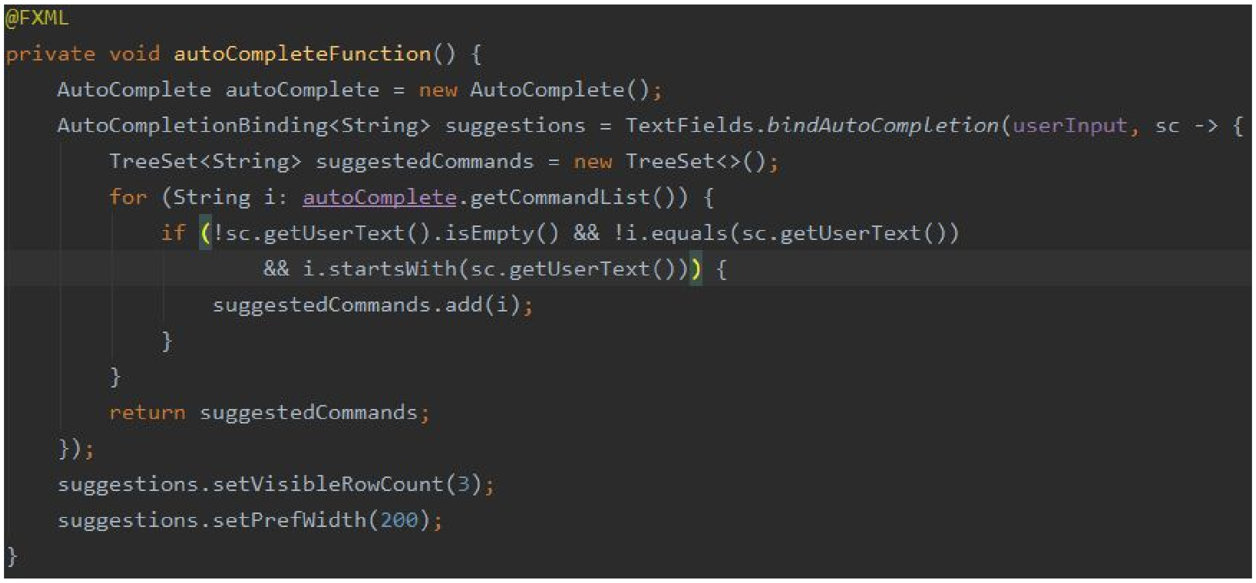
1. Suggestions of commands function
2. Memorising previous commands function

3.8.1. Suggestion of Commands Function

The Suggestions of Commands 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 **TestFields** creates a new auto-completion binding between the given textField and the given suggestion provider.

The exact implementation is shown below:


```
@FXML
private void autoCompleteFunction() {
    AutoComplete autoComplete = new AutoComplete();
    AutoCompletionBinding<String> suggestions = TextFields.bindAutoCompletion(userInput, sc -> {
        TreeSet<String> suggestedCommands = new TreeSet<>();
        for (String i: autoComplete.getCommandList()) {
            if (!sc.getUserText().isEmpty() && !i.equals(sc.getUserText())
                    && i.startsWith(sc.getUserText())) {
                suggestedCommands.add(i);
            }
        }
        return suggestedCommands;
    });
    suggestions.setVisibleRowCount(3);
    suggestions.setPrefWidth(200);
}
```

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.

3.8.2. Memorising Previous Commands Function

The Memorising of Previous Commands Functions are facilitated by the MainWindow from the UI Component.

3.8.3. Design Considerations

- **Auto-Completion Binding**

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

  Pros: Suggested Commands are accurate.

Cons: If the list gets too long, users are required to look through the list

Alternative 2: Suggesting any commands that contains user input

Pros: Displays more suggestions.

Cons: Unnecessary and Inaccurate suggestions tends to appear.

3.10. Statistics Feature

3.9.1 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.

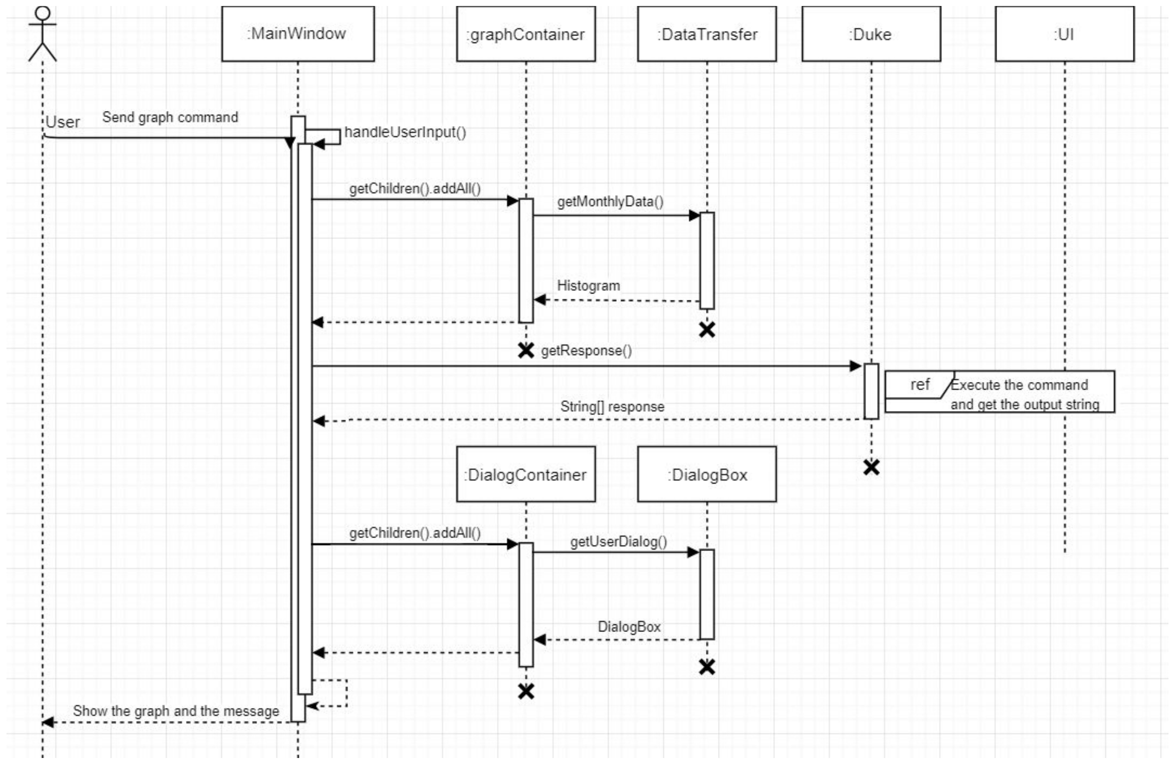
Step 3. After presenting the graph, the input string will be handled by Duke which will return a string as a response to the dialogue container in MainWindow.

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.



3.9.2 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 items in the database

3.10.1 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.

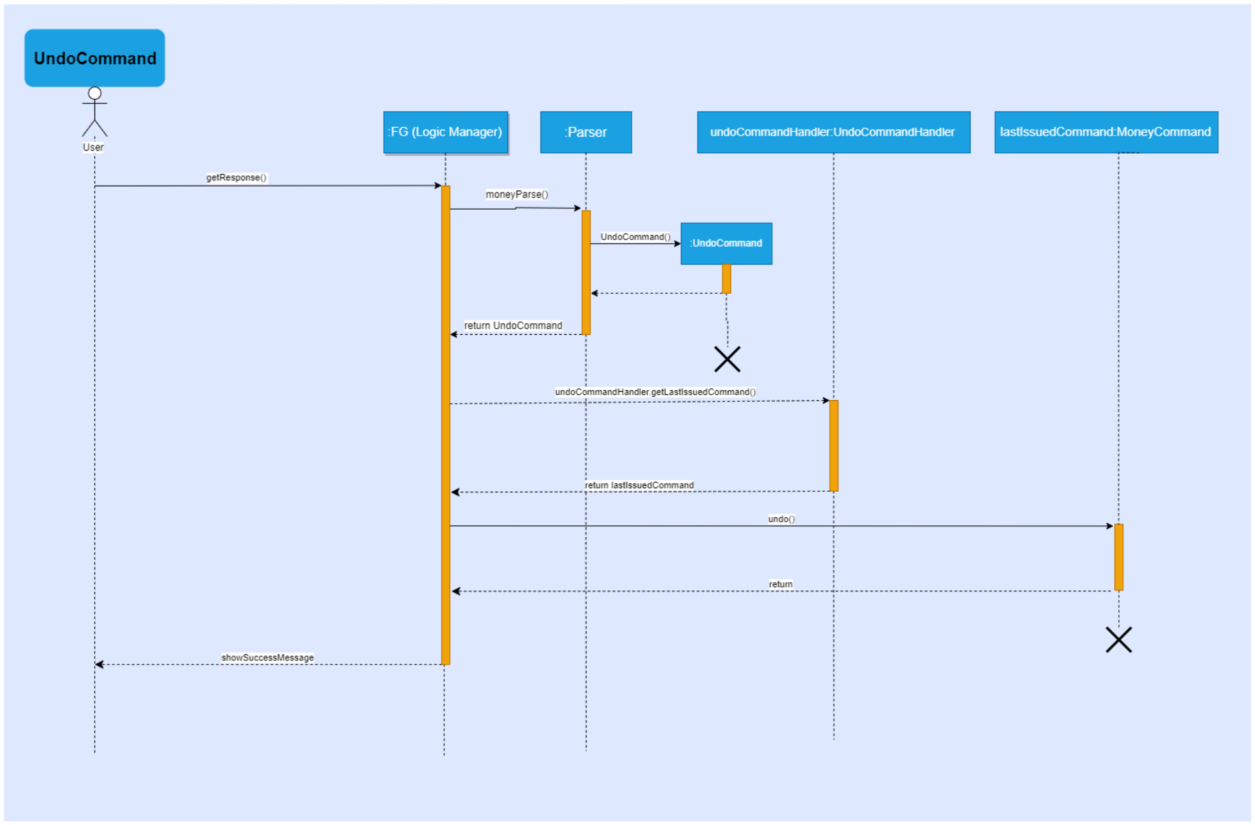
v. UI prints a success message.

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:



3.10.2 Design Considerations

- **Undoing Multiple Commands**

The program is allowed to 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. Logging

3.13. Configuration

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 Ghosts(FG)
- Actor: User
- Use Case: Check account balance
    1. User opens the application
    2. UI prompts for PIN
    3. User enters PIN
    4. FG greets and prompts an action list
    5. User enters the command to check the current balance
    6. UI prompts the related information about the balance
    7. FG waits for the user’s next command

  Use Case Ends.

- Extensions
    1. Incorrect PIN entered
        - FG displays an error message
        - Use Case continues at step 1
    2. The given command is invalid
        - FG displays an error message
        - Use Case continues at step 4

Use Case 2

- System: Financial Ghosts(FG)
- Actor: User
- Use Case: Record Finances
    1. User logs in the account
    2. UI prompts for PIN
    3. User enters PIN
    4. FG greets and prompts an action list
    5. User enters the command to declare the amount of money as income/expenditure
    6. UI prompts a “successfully recorded” message
    7. FG waits for the user’s next command

  Use Case Ends.

- Extensions
    1. Incorrect PIN entered
        - FG displays error message
        - Use Case continues at step 1

2. The given command is invalid
    - FG displays error message
    - Use Case continues at step 4
3. The given income has invalid characters in it
    - FG displays error message
    - Use Case continues at step 5

Use Case 3

- System: Financial Ghosts(FG)
- Actor: User
- Use Case: View Financial Statistics
    1. User opens the application
    2. UI prompts for PIN
    3. User enters PIN
    4. FG greets and prompts an action list
    5. User enters the command to view the financial statistics
    6. UI displays the relevant statistics to the user
    7. FG waits for the user’s next command

  Use Case Ends.

- Extensions
    1. Incorrect PIN entered
        - FG displays an error message
        - Use Case continues at step 1
    2. The given command is invalid
        - FG displays an error message
        - Use Case continues at step 4

Use Case 4

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

Use Case Ends.

- Extensions
    1. Incorrect PIN entered
        - FG displays an error message
        - Use Case continues at step 1
    2. The given command is invalid
        - FG displays an error message
        - Use Case continues at step 4

Use Case 5

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

    Use Case Ends.

- Extensions
    1. Incorrect PIN entered
        - FG displays an error message
        - Use Case continues at step 1
    2. The given command is invalid
        - FG displays an error message
        - Use Case continues at step 4

Use Case 6

- System: Financial Ghosts(FG)
- Actor: User
- Use Case: get recommended savings to reach set goals
    1. User opens the application
    2. UI prompts for PIN
    3. User enters PIN
    4. FG greets and prompts an action list

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

Use Case Ends.

- Extensions
  1. Incorrect PIN entered
     - FG displays an error message
     - Use Case continues at step 1
  2. The given command is invalid
     - FG displays an error message
     - Use Case continues at step 4

Use Case 7

- System: Financial Ghosts(FG)
- Actor: User
- Use Case: Record Outstanding Loans
  1. User opens the application
  2. UI prompts for PIN
  3. User enters PIN
  4. FG greets and prompts an action list
  5. User enters the command to settle a loan that has been taken/given out from/to another party.
  6. FG clears the loan settled and adds to expenditure/income depending on loan being taken/given out from/to another party
  7. UI displays a “loan settled” message.
  8. FG waits for the user’s next command

Use Case Ends.

- Extensions
  1. Incorrect PIN entered
     - FG displays an error message
     - Use Case continues at step 1
  2. The given command is invalid
     - FG displays an error message

        - Use Case continues at step 4

Use Case 8

- System: Financial Ghosts(FG)
- Actor: User
- Use Case: Settle Outstanding Loans
    1. User opens the application
    2. UI prompts for PIN
    3. User enters PIN
    4. FG greets and prompts an action list
    5. User enters the command to add a loan that has been taken/given out from/to another party.
    6. UI displays a “successfully added” message.
    7. FG waits for the user’s next command

    Use Case Ends.

- Extensions
    1. Incorrect PIN entered
        - FG displays an error message
        - Use Case continues at step 1
    2. The given command is invalid
        - FG displays an error message
        - Use Case continues at step 4

Use Case 9

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

    Use Case Ends.

- Extensions

1. Incorrect PIN entered
    - FG displays an error message
    - Use Case continues at step 1
2. The given command is invalid
    - FG displays an error message
    - Use Case continues at step 4

3. Auto Deductions

    - Monthly deductions occurs at specific date
    - Use Case continues at step 1

Use Case 10

- System: Financial Ghosts(FG)
- Actor: User
- Use Case: Archive Financial Statements
    1. User opens the application
    2. UI prompts for PIN
    3. User enters PIN
    4. FG greets and prompts an action list
    5. User enters the command to archive a specified month's financial statements.
    6. UI displays a "successfully archived" message.
    7. FG waits for the user's next command

    Use Case Ends.

- Extensions
    1. Incorrect PIN entered
        - FG displays an error message
        - Use Case continues at step 1
    2. The given command is invalid
        - FG displays an error message
        - Use Case continues at step 4
    3. FG fails to write to the archive
        - FG displays an error message

    Use Case Ends.

Use Case 11

- System: Financial Ghosts(FG)
- Actor: User

- Use Case: Provides Assistance to the User
    1. User opens the application
    2. UI prompts for PIN
    3. User enters PIN
    4. FG greets and prompts an action list
    5. User enters the command to get a detailed description of specified action.
    6. UI displays a detailed description of the action.
    7. FG waits for the user’s next command

  Use Case Ends.

- Extensions
    1. Incorrect PIN entered
        - FG displays an error message
        - Use Case continues at step 1
    2. The given command is invalid
        - FG displays an error message
        - Use Case continues at step 4

Use Case 12

- System: Financial Ghosts(FG)
- Actor: User
- Use Case: Search for items
    1. User opens the application
    2. UI prompts for PIN
    3. User enters PIN
    4. FG greets and prompts an action list
    5. User enters the command to get search for an item using a keyword by specifying the desired field of search.
    6. UI displays a list of the search results.
    7. FG waits for the user’s next command

  Use Case Ends.

- Extensions
    1. Incorrect PIN entered
        - FG displays an error message
        - Use Case continues at step 1
    2. The given command is invalid
        - FG displays an error message

        - Use Case continues at step 4

Use Case 13

System: Financial Ghosts(FG)

- Actor: User
- Use Case: Check Installments
    1. User opens the application
    2. UI prompts for PIN
    3. User enters PIN
    4. FG greets and prompts an action list
    5. User enters the command to check the instalments that are in the system.
    6. UI displays all of User’s instalments to the output.
    7. FG waits for the user’s next command.

  Use Case Ends.

- Extensions
    1. Incorrect PIN entered
        - FG displays an error message
        - Use Case continues at step 1
    2. The given command is invalid
        - FG displays an error message
        - Use Case continues at step 4

Use Case 14

- System: Financial Ghosts(FG)
- Actor: User
- Use Case: Pay Outstanding Bills
    1. User opens the application
    2. UI prompts for PIN
    3. User enters PIN
    4. FG greets and reminds user of outstanding bills due in the coming week or sooner/overdue
    5. User keys in bills that he/she has already paid.
    6. FG removes reminder for the bill.

  Use Case Ends.

- Extensions

1. The given command is invalid
    - FG displays an error message
    - Use Case continues at step 4

Use Case 15

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

  Use Case Ends.

- Extensions
    1. Incorrect PIN entered
        - FG displays an error message
        - Use Case continues at step 1
    2. The given command is invalid
        - FG displays an error message
        - Use Case continues at step 4
    3. FG fails to write to the archive
        - FG displays an error message

  Use Case Ends.

Use Case 16

- System: Financial Ghosts(FG)
- Actor: User
- Use Case: Input add commands in stages
    1. User opens the application
    2. UI prompts for PIN
    3. User enters PIN
    4. FG greets and prompts an action list

5. User enters the type of add command to their accounts
6. UI prompts the user for the next data to add for the respective command with proper syntax guide until all data required is inputted
7. UI displays a “successfully recorded” message.
8. FG waits for the user’s next command

Use Case Ends.

- Extensions
  1. Incorrect PIN entered
     - FG displays an error message
     - Use Case continues at step 1
  2. The given command is invalid
     - FG displays an error message
     - Use Case continues at step 4
  3. The given input for add command data is invalid
     - FG displays an error message
     - Use Case continues at step 6

Use Case Ends.

Use Case 17

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

Use Case Ends.

- Extensions
  1. Incorrect PIN entered
     - FG displays error message
     - Use Case continues at step 1

```
    2. 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

FG - Financial Ghost: The name of our application

UI - User Interface: Things that the user will interact while using the application