Duke$$$

An NUS Software Development Project

1. Table of Contents

2. Introduction

Duke$$$ is a one-of-a-kind desktop application that aims to provide financial tracking for international university students who are on semester exchange to NUS. Built using the Java language, DUKE$$$ is developed to have a Graphic User Interface (GUI), but uses Command Line Interface (CLI) for all user interactions.

2.1. Purpose

This document describes the overall architecture and implementations of DUKE$$$.

2.2. Audience

The intended audience for this document are users and developers of DUKE$$$, and any general audience who would like to read up on the design and implementations made on DUKE$$$. A basic understanding of Java and Object-Oriented Programming (OOP) is recommended.

2.3. Document Organization

Section	Purpose
Section 3. Setting Up	To guide users to set up DUKE$$$
Section 4. Design	To introduce the system architecture of DUKE$$$ and describe each component layer of the program
Section 5. Implementation	To describe how the key features of DUKE$$$ was implemented
Section 6. Documentation	To describe how the documentation of the developer guide was carried out

2.4. User Guide

Refer here.

3. Setting Up

This section describes the procedures for setting up Duke$$$

3.1. Prerequisites

1. JDK 11 or later
2. IntelliJ IDE

3.2. Setting up the project in your own Desktop

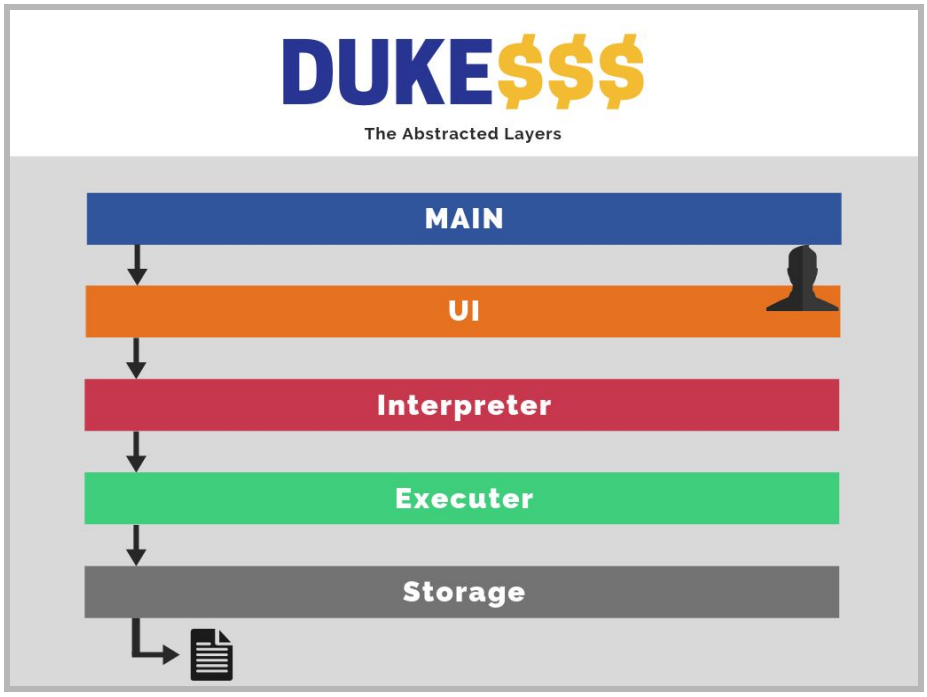
1. Fork this repo and clone the forked repository on to your desktop.
2. Open IntelliJ
3. Set up the correct JDK version for Gradle.
    I. Click Configure --> Project Defaults --> Project Structure
    II. Click New and locate the directory of the JDK.
4. Select Import Project.
5. Find the build.gradle file and select it.
6. Select Open as Project.
7. Click OK to accept the default settings.
8. Run the class named Duke.
9. You are now able to execute the commands which can be seen if you type help on the Command Line Interface (CLI)
10. You may wish to run the tests optionally to ensure successful build to verify setup.

4. Design

Given the urgency of the project, the team opted to approach the project by implementing an agile design that could adapt to any unexpected requirements imposed by the `Contractor`. As the project is expected to grow beyond the term of the present team's period of service, a multi-level design was deemed necessary to ensure scalability of the product.

4.1. Architecture

To facilitate future handovers to other incoming teams, the team adopted the n-tier architectural style which abstracts the workings of the application into separate layers that can each be understood separately. This is in line with the need for scalability and also allows teams of developers to focus on improving a single layer without drastically disturbing the workings of other layers. The diagram below illustrates the high-level design of the application:



As seen in the diagram above, the `User` primarily interacts with the `Ui` Layer which in turn, only interacts with the layers adjacent to it. The sections below explore in greater detail the individual structure of each layer:

4.2. Main Layer

The `Main` Layer contains a single class known as `Duke`. This is the main application layer that loads on the user's computer and handles the authentication and account management of all users.

`Duke`'s main functions are:

- Authenticating users who have previously created an account in Duke$$$
- Creating new accounts for new users
- Providing a link to the User Guide

4.3. Ui Layer

The `Ui` Layer is the primary layer that the `User` interacts with. As such, classes in this layer focuses on ensuring a good user experience and displaying any data requested by the `User` . It consists of two main classes: `TextUi` and `GraphicalUi`.

`TextUi` handles:

- Displaying text that the `User` inputs
- Displaying suggestions based on `User` input

`GraphicalUi` handles:

- Displaying graphs and charts representing users financial data
- Managing the `User's` preferred color themes

4.4. Interpreter Layer

The `Interpreter` Layer is responsible for interpreting any input provided by the `User` . This includes parsing for pre-set commands as well as adjusting to any user-defined shortcuts. It consists of solely the `Parser` Class.

`Parser`'s main functionality include parsing for:

- Commands
- Dates

4.5. Executor Layer

The `Executor` Layer receives instructions from the `Interpreter` Layer and executes the `Command` required of it. Comprising of two sub-layers, `Command` Layer and `Task` Layer, the `Executor` Layer is responsible for the following actions:

- `CommandGetMonthlySpending` - Outputs the total spending for a given month
- `CommandGetYearlySpending` - Outputs the total spending for a given year
- `CommandAddReceipt` - Template for receipt adding commands
- `CommandAddSpendingReceipt` - Creates a spending receipt
- `CommandAddIncomeReceipt` - Creates an income receipt
- `CommandNewTask` - Creates and stores a new `Task`
- `CommandMarkDone` - Marks an existing `Task` as 'done'
- `CommandReminder` - Generates a reminder based on an existing `Task`
- `CommandSchedule` - Displays the schedule based on a given date
- `CommandConvert` - Converts currency between countries
- `CommandWeather` - Display real time weather information based on period requested
- `CommandQueue` - Creates and queues a new `Task` behind an existing one
- `CommandDelete` - Deletes a certain data entry based on index
- `CommandFind` - Locates and displays a certain data entry
- `CommandList` - Lists all the data stored by the `User`
- `CommandBlank` - Executes nothing
- `CommandError` - Throws an Error
- `CommandSave` - Saves the `User`'s Data
- `CommandLoad` - Loads the `User`'s Data
- `CommandBye` - Logs out the `User`
- `CommandlistTag` - Lists all the receipts corresponding to the tag entered by `User`

When executed, `CommandNewTask` passes on a request from the `Command` sublayer to the `Task` sublayer which contains the following classes:

- `Task` - Defines the abstract class for all `Tasks`
- `TaskType` - Defines the enumerations used to identify different `Tasks`
- `TaskList` - Tracks the `Tasks` created by the `User`
- `Deadline` - Defines the functionality of the `Task` subclass 'Deadline'
- `Event` - Defines the functionality of the `Task` subclass 'Event'
- `ToDo` - Defines the functionality of the `Task` subclass 'ToDo'
- `FixedDuration` - Defines the functionality of the `Task` subclass 'FixedDuration'
- `Recur` - Defines the functionality of the Task subclass 'Recurring'

4.6. Storage Layer

The `Storage` Layer is responsible for saving and loading the `User`'s data. Data from the `TaskList` will be encoded into strings that are stored in a .txt file. These strings are later decoded by the `Storage` Layer before populating the `Ui` . This layer consists of `StorageTask` and `StorageWallet` Classes.

`Storage`'s main functionality include:

- Encoding data
- Decoding previously stored data
- Saving data into a .txt file
- Loading data from a .txt file

5. Implementation

This section describes how certain features are implemented and function.

5.1. Interpreting a User's Input

Interpretation of the input is done in the `Interpreter` Layer, specifically by the `Parser` Class. As user-experience is a priority, the `Parser` must be able to correctly interpret what the `User` is trying to do even through minor typing mistakes. Nonetheless, a certain structure must be adhered to ensure the reliability of the interpreter.

5.1.1. Instruction Structure

Instructions must generally follow the following structure:

`command` `primaryInput` /`flag` `flagDetails` ...

5.1.2. Instruction Inputs

Each variable used in the structure above is a placeholder for a particular kind of input. These are:

`command`

This placeholder specifies what type of `Command` to execute.

`primaryInput`

This can refer to a number of things. For example, if the intention was to execute `CommandNewTask`, `primaryInput` would then be the placeholder for the title of the task. Comparatively, if `CommandDelete` was specified, `primaryInput` will instead refer to the index of the task to be deleted.

/`flag`

The `flag` placeholder must be attached to the forward slash. This '/' is a mandatory delimiter. `flag` lets the `User` specify a specific case of the `Command` in question.

`flagDetails`

The `flagDetails` placeholder is interpreted immediately after the `flag` and continues until another `flag` is found or the end of the input is reached. `flagDetails` can refer to a number of things and, similar to `primaryInput`, depends on what `flag` is specified. For example, if the input was "*Event My Event /on* `flagDetails`", then, `flagDetails` should refer to a date.

5.1.3. Main Parser Methods

Given the Instruction Inputs above, the `Parser` Class in the `Interpreter` Layer has three main methods to extract these inputs for the convenience of the `Developer` . These methods are:

- ***parseForCommandType***

  Returns a `CommandType` enumeration corresponding to `command`.

- ***parseForPrimaryInput***

  Returns the `primaryInput` as a `String`.

- ***parseForFlag***

  This method returns the `flagDetails` as a `String` given a specified `flag`. This allows the `Developer` the flexibility to implement any number of `flags` with any names to required in a particular `Command`.

5.1.4. Design Considerations

Delimiters

Although the instruction structure above displays 4 delimiters (3 whitespaces and 1 forward slash), in actuality, only the forward slash for each `flag` is mandatory.

Initial Design

Initially, all 4 delimiters were required to parse the user input using commands such as *string.split('delimiter')*. However, the team quickly realized the frustration that occured after typing in a long command only to have it fail due to a missing whitespace. As such, to improve the general user experience, the number of delimiters required had to be reduced.

Alternative Design

Instead of using *string.split('delimiter')*, the team opted to parse the input using a combination of *string.indexOf('delimiter')* and a minimum index loop. This allows the Parser to take advantage of the order that the Instruction Structure requires as the `command` placeholder is always in the first part of the input. The advantage of this approach is that the input "*Delete5*" still deletes the 5th-indexed task in the list even through the lack of any whitespace. In fact, the absence/presence of whitespace at any part of the user input will not affect the interpretation of the input. Furthermore, this design allows `flags` to be rearranged in any order to suit the `User`'s preference. Given that the total commands are limited and that user inputs are, on average, not very long, the increase in execution time is non-consequential and the `User` is granted an easier time inputting commands through the Command Line Interface (CLI).

5.2. TaskList

`TaskList` is the Class that contains the `ArrayList` of `Task` Objects as well as other methods relevant to keeping track and updating this list.

5.3. Expenditure

The **expenditure** feature of DUKE$$$ is facilitated by the `CommandGetSpendingByDay`, `CommandGetSpendingByWeek`, `CommandGetSpendingByMonth`, `CommandSpendingByYear` for the day, week, month, year respectively. All the above commands inherits from the parent class `Command` and overrides the execution method to output the total amount of money spent by the user, categorically.

The general flow of the logic constitutes of the Receipts that are being input by the user tracked by the `ReceiptTracker`. Following that, when the user inputs the command to know the total amount of spending for the month, the logic finds all the receipts that correspond to the month's input from the user and outputs the amount.

Given below is a more detailed explanation of the feature and a break down of how it works in given example scenarios.

5.3.1. Daily Expenditure

{In Progress...}

5.3.2. Weekly Expenditure

{In Progress...}

5.3.3. Monthly Expenditure

Step 1: `User` inputs the command to get the total expenditure for the month in the following format through the Command Line Interface:

```
expendedmonth september /year 2019
```

Step 2: `Ui` captures the `userInput` and sends it to the `Parser`, which will then parse the command to identify its `CommandType`. Based on the sample `userInput` specified above,

`CommandType` - **`EXPENDEDMONTH`**

Step 3: Since the `CommandType` is **`EXPENDEDMONTH`**, the program invokes `CommandGetSprendingByMonth`.

Step 4: `CommandGetSprendingByMonth` is instantiated with the `userInput` passed as an argument. The execute method called the other methods associated with the calculation of the total expenditure.

Firstly, we parse the input through 2 methods:

> `Parser.parseForPrimaryInput` -- this will identify the month
>
> `Parser.parseForFlag` -- this method is invoked to parse for the flag "year" in order to obtain the year of the month that the expenditure detail is required.

Based on the sample userInput specified above,

> `PrimaryInput` -- **`september`** (month)
>
> `flag1 details` -- **`2019`** (year)

Secondly, we invoke the `monthStrToInt` method to convert the month from a string to its corresponding numerical value

September - **`9`** (Numerical value of the month)

Step 5: Thirdly, we pass the `PrimaryInput` and `flag1 details` to the method `findReceiptByMonthYear`. The program runs through all the receipts and takes all those which corresponds to the month and year details from the `PrimaryInput` and `flag1 details` and stores it into an ArrayList. Then `findReceiptByMonthYear` returns this ArrayList.

Step 6: Finally the monetary value of the total expenditure is accessed through the method `getTotalCashSpent`. And the value is printed out.

5.3.4. Yearly Expenditure

Step 1: User inputs the command to get the total expenditure for the month in the following format through the Command Line Interface:

`expendedyear 2019`

Step 2: `Ui` captures the `userInput` and sends it to the `Parser,` which will then parse the command to identify its `CommandType`. Based on the sample `userInput` specified above,

`CommandType` **`- EXPENDEDYEAR`**

Step 3: Since the `CommandType` is **`EXPENDEDYEAR`**, the program invokes `CommandGetSprendingByYear`.

Step 4: `CommandGetSprendingByYear` is instantiated with the `userInput` passed as an argument. The execute method called the other methods associated with the calculation of the total expenditure.

Firstly, we parse the input through 1 method:

`Parser.parseForPrimaryInput` -- this will identify the year

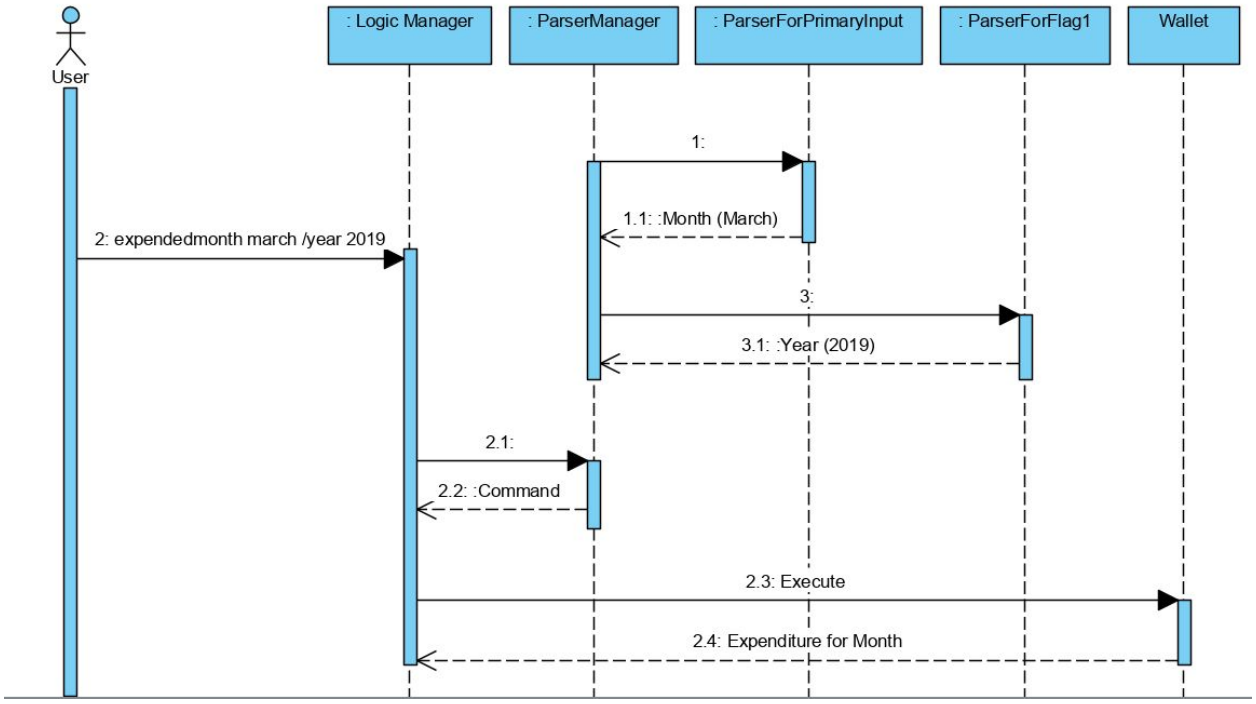
Based on the sample userInput specified above,

`PrimaryInput` -- `2019` (year)

Step 5: Secondly, we pass the `PrimaryInput` to the method `findReceiptByMonthYear`. The program runs through all the receipts and takes all those which corresponds to the month and year details from the `PrimaryInput` and stores it into an ArrayList. Then `findReceiptByMonthYear` returns this ArrayList.

Step 6: Finally the monetary value of the total expenditure is accessed through the method `getTotalCashSpent`. And the value is printed out.

The diagram below shows the sequence of the flow of events for this feature:



5.3.1. Sequence diagram for expenditure events

The future improvements of the feature include:

- Separating the receipts according to weekly expenses
- Separating the receipts according to daily expenses

This allows the user to get a more detailed view of his/her expenditure and is able to keep a closer track of all of them.

5.4. Addition of new receipts into a User's wallet

The **addition of new receipts into the User's wallet** is facilitated by `CommandAddSpendingReceipt` and `CommandAddIncomeReceipt`, both extended from `CommandAddReceipt`. The Wallet execution is used to add new receipt input from the user to the existing database. `CommandAddSpendingReceipt` and `CommandAddIncomeReceipt` adds a new receipt for spending and income respectively. `User` can indicate their date of acquisition of the receipts or label tags for easier classification of their spending and income. If

not indicated, the date of the receipt is primarily set as the date of input. The new receipts are added into the User's Wallet object though `Wallet.AddReceipt`.

Step 1: `User` inputs the receipt command, cash value, date and tags in the following format into the Command Line Interface.

a. For income: `CommandType` `-- IN`

```
In $200.0 /date 2019-08-30 /tags angbao, parents
In $3
```

b. For spending: `CommandType` `-- OUT`

```
Out $2.00 /date 2019-12-10 /tags icecream
Out $15.0 /date 2018-08-25
```

Step 2: The `Parser` parses the first word in the userInput string to determine the `CommandType` of the input and determine the command to invoke.

`CommandType` `-` `IN` calls `CommandAddIncomeReceipt`

`CommandType` `-` `OUT` invokes `CommandAddSpendingReceipt`.

Step 3: Parsers parse the necessary information from the String though following methods:

`extractIncome` -- Parses the cash value as Double

`extractDate` -- Parses the date as Date

`extractTags` -- Parses the tags as a String array

The parsed parameters are then passed to the invoked child command, according to CommandType.

Step 4: Objects for receipts are instantiated with the passed parameters.

`CommandAddIncomeReceipt` -- a new `IncomeReceipt` object is instantiated

`CommandAddSpendingReceipt` -- a new `Receipt` object is instantiated

The new `IncomeReceipt` or `Receipt` object is then added to the Wallet Object through `Wallet.addReceipt`.

Step 5: The `User` is notified of the creation of the new `IncomeReceipt` or `Receipt` object through a message shown in the UI.

Further improvements for this feature includes:

- Allowing User to create new Receipts in other currencies
- Allowing the User to delete Receipts

5.5. Currency Converter

The **currency conversion** feature of DUKE$$$ is facilitated by `CommandConvert`. It inherits from the parent class `Command` and overrides the execution method to output the user's choice of desired currency and converted amount, along with the exchange rate that has been used.

The implementation of this feature requires the use of an API which provides real-time exchange rates between EUR and other countries in json. We convert json into a java string and then parse to obtain the desired exchange rates for conversion. This currency conversion is a two step process :

I. Converting user entered amount from base currency into EUR
II. Converting the amount in EUR into required currency entered by user

However, if the required or base currency is EUR, the logic of the process is simplified.

The following steps describe the implementation of the logic in greater detail:

Step 1: `User` inputs the amount for conversion, base currency and the currency required in the following format through the Command Line Interface :

convert 2500 /from USD /to INR

Step 2 : `Ui` captures the `userInput` and sends it to the `Parser`, which will then parse the command to identify its `CommandType`. Based on the sample `userInput` specified above,

`CommandType` -- **`CONVERT`**

Step 3: Since the `CommandType` is **`CONVERT`**, the program invokes `CommandConvert`

Step 4: `CommandConvert` is instantiated with the `userInput` passed as an argument. The execute method calls the other methods associated with currency conversion. Firstly, we parse the input through 2 methods :

`Parser.parseForPrimaryInput` -- this will identify the amount for conversion

`Parser.parseForFlag` -- this method is invoked twice to parse for the two flags “from” and “to” in order to obtain the base currency and currency required

Based on the sample userInput specified above,

`PrimaryInput` -- **`2500`** (amount)

`flag1 details` -- **`USD`** (base currency)

`flag2 details` -- **`INR`** (currency required)

Step 5: Secondly , we pass the `flag1 details` and `flag2 details` to the method `consultTheCurrencyApi` which will make an api call and return a string containing the json with the necessary exchange rates, with EUR as the base currency. If either of `flag1 details` or `flag2 details` is EUR , the link for the API call is changed accordingly using the method `generateApiUrl`.

Step 6: Thereafter we make use of `gson` by google to read the json string and identify the two exchange rates between countries required and EUR through the method `deriveExchangeRateFromJson` . Using that, we convert the amount from the base currency to EUR which inturn is converted into the required currency.

Step 7 : We output the desired currency, amount and the exchange rate used through the `Ui`

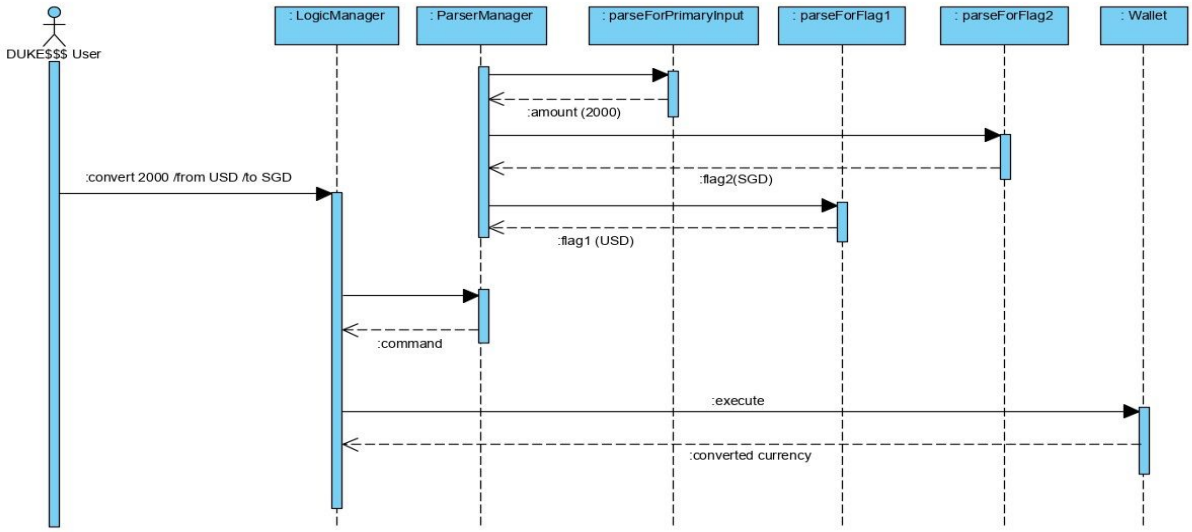


Figure 5.5.1. Sequence Diagram when converting currency between countries

The diagram above shows the sequence of processes which occur behind the logic described under 5.5 Currency Converter.

5.6. Weather Display

The **weather display** feature of DUKE$$$ is facilitated by `CommandWeather`. It inherits from the parent class, `Command` and overrides the execution method to output the user's choice of desired weather update.

The implementation of this feature requires the use of an API which provides real-time weather status for the next four days in Singapore. We convert json into a java string and then parse to obtain the required fields which will be stored in a nested hashmap. Depending upon the user's request we choose `until` which day's data to be displayed in real-time.

The following describes the implementation of the logic in greater detail :

Step 1: User inputs the day `until` which he requires weather information in the following format through the Command Line Interface :

weather /until tomorrow

Step 2 : `Ui` captures the `userInput` and sends it to the `Parser`, which will then parse the command to identify its `CommandType`. Based on the sample `userInput` specified above,

`CommandType` -- **`WEATHER`**

Step 3: Since the `CommandType` is **`WEATHER`**, the program invokes `CommandWeather`

Step 4: `CommandWeather` is instantiated with the `userInput` passed as an argument. The execute method calls the method to fetch the required weather data for display. Firstly, we parse the input :

`Parser.parseForFlag` -- this method is invoked to parse for the flag “until” in order to obtain the time period until which the user requests for weather data.

Based on the sample userInput specified above,

`flag details` -- **`tomorrow`**

Step 5: Secondly , we call the method `consultWeatherApi` which will make an api call and return a string containing the json with all the weather data for the next four days including the present instant.

Step 6: Thereafter through the method `storeWeatherDataFromJson` we make use of `gson` by google to read the json string and identify the required fields for user display. We store them in a nested hashmap.

Step 7 : The `flag details` is actually the day until which user queries for weather data. Hence we use the method `getLengthOfHashMapToPrint` to determine the day until we have to print the stored weather data.

Step 8 : We output the desired weather information through the `Ui`

The diagram below shows the sequence of processes which occur behind the logic described :

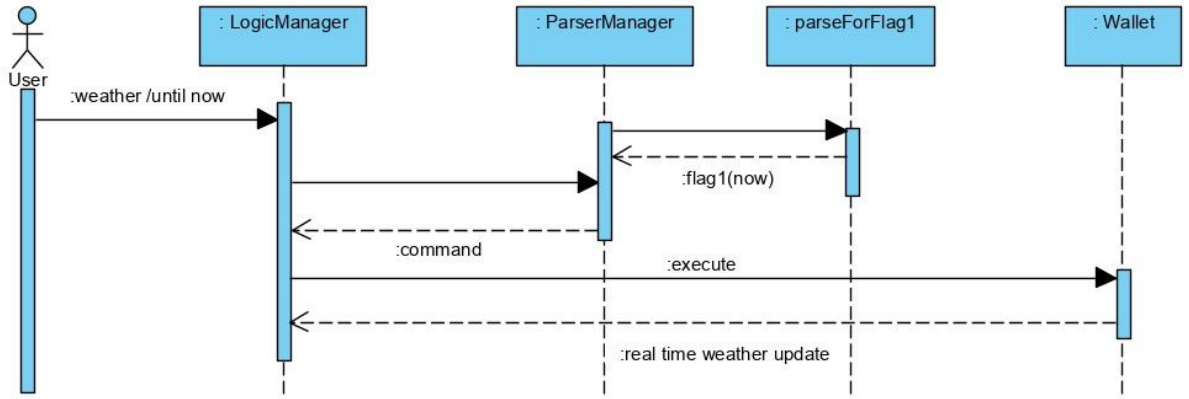


Figure 5.6.1. Sequence Diagram when displaying user requested weather data

5.7. Listing based on Tag

Listing the expenditure based on tag feature

The **listTag** feature is facilitated by CommandListTag. It extends from CommandList (Which extends from command) and uses the wallet execution to output the total expenditure based on the tag and the respective tag receipts.

The general flow of logic constitutes the receipts that are being inputted by the user tracked by the ReceiptTracker. The user inputs the command taglist followed by the tag(Eg Food/Transport) and the logic finds all the receipts that contain the tag and outputs the list along with the total expenditure on that particular tag.

Given below is a more detailed explanation.

Step 1: User inputs the taglist command and the tag to the Command Line Interface in the following format :

```
taglist food
```

Step 2 : `Ui` captures the `userInput` and sends it to the `Parser`, which will then parse the command to identify its `CommandType`. Based on the sample `userInput` specified above,

`CommandType` -- taglist

Step 3: Since the `CommandType` is TAGLIST, the program invokes `CommandListtag`

Step 4: `CommandListtag` is instantiated with the `userInput` passed as an argument. Firstly, we parse the input :

`Parser.parseForPrimaryInput` -- this method is invoked to parse for the tag that is inputted after the command

Based on the sample userInput specified above,

`Primary input details` -- food

Step 5: When the application is launched, Wallet and ReceiptTracker are initialised. It will have the previously saved receipts given by the user and would display the total amount of expenditure and balance of the account.

Step 6: Within the wallet class the tag name is used to find all the corresponding tags in the list using the method findReceiptsByTag. This returns a list containing all the receipts with the specific tag. It is stored in the ReceiptTracker object called 'taggedreceipts'.

Step 7: The monetary value of the total expenditure for that specific tag is accessed through the method getTotalCashSpent. And the value is printed out.

Step 8: We output the desired list of receipts corresponding to the tag

The diagram below gives a more detailed explanation about the sequence

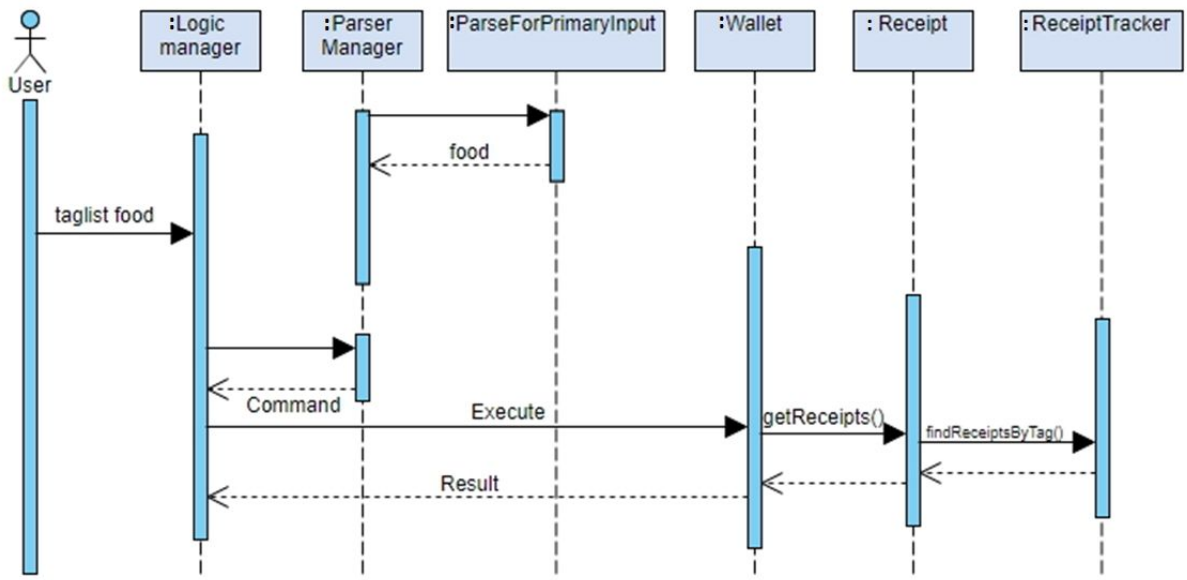


Figure 5.6.2. Sequence Diagram when listing receipts by Tag

The future improvements of this feature include

- Adding depth to this feature by adding support for the Duke to track what the user asked to filter by.

Appendix A. Product Scope

The scope of DUKE$$$ is to meet the needs of exchange students in Singapore who have the following requirements :

- Needs assistance managing their expenses while in Singapore
- Finds CLI applications more alluring than GUI applications
- Prefers Desktop applications over the Web or Mobile applications
- Wishes to plan his or her travel around Singapore based on weather forecast
- Wishes to budget and convert currencies for travel
- Able to use the application with Internet access for full access to all features

However the scope of the product is not restricted to exchange students albeit it is customised to suit their needs. This application is still very much applicable to any general user , comfortable with the Command Line Interface and wishes to keep track of their expenses through meaningful visualisation of expenditure statistics.

Value Proposition

Manage , review and maintain financial expenses and income receipts using CLI with data visualisation on the GUI.

Appendix B. User Stories

This section lists the user stories that the developer team of DUKE$$$ has ideated. These user stories were used to narrow down on the required features for a useful and functioning desktop application that serves as a financial tracker for exchange students here in Singapore.

The user stories are categorized into different priorities for implementation:

- High (must have) --- ***
- Medium (nice to have) --- **
- Low (not necessary but applicable) --- *

Priority	As a ...	I want to ...	So that I can ...
***	New user	See usage instructions	Refer to instructions when I forget how to use the App
***	User	See a dashboard with all my budget plans, current total expenditure and available balance	Be updated of my financial status
***	Student	Calculate my expenses	Manage my finances better
***	Exchange Student	Convert my home currency into any currency around the world	Convert SGD into other currencies should i be travelling over the weekends
***	User	See a data chart which shows the comparison of expenditure per category	Track my expenditures and remind myself to spend less if i have to
***	Exchange Student	See live weather forecast	Plan my travel around Singapore

***	User	See my daily/weekly/monthly/ yearly expenditure	Keep track of my expenses
***	User	Add tags to the various expenditures	Sort by expenses and see where i am spending more
**	User	Add income which i have received	Update my total balance if i receive cash inflow
**	User	Interact with the app through a graphical user interface.	Interact with the application with more ease
*	Exchange Student	Connect with fellow exchange students based on: - Country of origin - Gender - Period of Exchange - Course of study - Interest groups This need not be through a chat interface.	Make my own group of friends for support network
*	Exchange Student	Interact with fellow exchange students through a real time chat interface	Form my support network while away from home
*	Millennial	Have different view mode (night/day)	Customise my App to my liking
*	Exchange Student	Have a list of travel itineraries in Singapore and the local delights.	Have an e- travel brochure which I can refer to that guides me when i want to travel or order food.

Appendix C. Use Cases

This section describes the Use Cases for some of the features implemented in DUKE$$$.

{In Progress….}

Use case 1: Adding a Spending Receipt

- **MSS:**
    1. User inputs …. Command with necessary arguments.
    2. Duke$$$ ……. *{what it does}*
    3. Use case ends.

- **Extensions**
    - 1a. Duke$$$ detects arguments are in the wrong order.
        - 1a.1. Duke$$$ outputs an error message
        - Use case ends.
    - 1b. Duke$$$ detects missing required arguments in the given user input.
        - 1b.1. Duke$$$ outputs an error message.
        - Use case ends.

Use case 2: Converting Currency

- **MSS:**
    1. User inputs `convert` Command with necessary arguments.
    2. Duke$$$ converts currency from the base currency to the required currency.
    3. Use case ends.

- **Extensions**

- 1a. Duke$$$ detects arguments are in the wrong order.
    - 1a.1. Duke$$$ outputs an error message
    - Use case ends.
- 1b. Duke$$$ detects missing required arguments in the given user input.
    - 1b.1. Duke$$$ outputs an error message.
    - Use case ends.

Use case 3: Displaying Weather

- **MSS:**
    1. User inputs `weather` Command with necessary arguments.
    2. Duke$$$ analyses the arguments and displays the weather data as per the period requested.
    3. Use case ends.

- **Extensions**
    - 1a. Duke$$$ detects arguments are in the wrong order.
        - 1a.1. Duke$$$ outputs an error message
        - Use case ends.
    - 1b. Duke$$$ detects missing required arguments in the given user input.
        - 1b.1. Duke$$$ outputs an error message.
        - Use case ends.

Use case 4: Display expenditure

- **MSS:**
    1. User inputs …. Command with necessary arguments.
    2. Duke$$$ ……. *{what it does}*
    3. Use case ends.

- **Extensions**
    - 1a. Duke$$$ detects arguments are in the wrong order.
        - 1a.1. Duke$$$ outputs an error message
        - Use case ends.
    - 1b. Duke$$$ detects missing required arguments in the given user input.
        - 1b.1. Duke$$$ outputs an error message.
        - Use case ends.

Use case 5: Display expenditure by Tags

- **MSS:**
    1. User inputs …. Command with necessary arguments.
    2. Duke$$$ ……. *{what it does}*
    3. Use case ends.

- **Extensions**
    - 1a. Duke$$$ detects arguments are in the wrong order.
        - 1a.1. Duke$$$ outputs an error message
        - Use case ends.
    - 1b. Duke$$$ detects missing required arguments in the given user input.
        - 1b.1. Duke$$$ outputs an error message.
        - Use case ends.

Use case 6: Adding an Income Receipt

- **MSS:**
    1. User inputs …. Command with necessary arguments.
    2. Duke$$$ ……. *{what it does}*
    3. Use case ends.

- **Extensions**
    - 1a. Duke$$$ detects arguments are in the wrong order.
        - 1a.1. Duke$$$ outputs an error message
        - Use case ends.
    - 1b. Duke$$$ detects missing required arguments in the given user input.
        - 1b.1. Duke$$$ outputs an error message.
        - Use case ends.

Appendix D. Non-functional Requirements

1. DUKE$$$ should work on any mainstream OS with Java 11 or higher installed.

2. DUKE$$$ should be able to hold up to 100 users database without having a noticeable deterioration in performance.

3. DUKE$$$ should have automated unit tests and an open source code.

4. DUKE$$$ should be able to work on both 32-bit and 64-bit environments.

5. The overall size of DUKE$$$ should not exceed 100MB.

6. DUKE$$$ should not contain any language deemed offensive to English speakers.

Appendix E. Glossary

Duke$$$

Financial Tracker

Mainstream OS

Windows, Linux, Unix, OS-X

Agile Design

Refers to an architectural design that evolves over time to take in new requirements.

N-tier Architectural Style

In the n-tier style, higher layers make use of services provided by lower layers. Lower layers are independent of higher layers

Abstraction

Abstraction refers to the act of representing essential features without including the background details or explanations

Layers

Layers refer to packages containing Java Classes which are arranged in levels according to the N-tier Architectural Style.